(2)

A RAND NOTE

A Table-Driven Approach to Fast Context-Free Parsing

James R. Kipps

December 1988

40 Years 1948-1988

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: N-2841-DARPA	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE and Subtitle: A Table-Driven Approach to Fast Context-Free Parsing	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): James R. Kipps	8. CONTRACT OR GRANT NUMBER(s): MDA903-85-C-0030
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The RAND Corporation, 1700 Main Street, Santa Monica, CA 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, Department of Defense, Arlington, VA 22209	12. REPORT DATE: December 1988
	13. NUMBER OF PAGES: 47
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parsers
Context Free Grammars
Algorithms
Tables (Data)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See reverse side

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This Note describes a parsing system developed for the ROSIE programming language and loosely based on Tomita's algorithm for general context-free parsing. Tomita's algorithm is unique in that it defines a bottom-up parser that uses an extended left-to-right (LR) parse table. While this algorithm does no better than the theoretical time bound of other general context-free parsing algorithms, its use of parse tables eliminates a component common to these algorithms. This makes Tomita's algorithm efficient enough for practical application. The parsing system described in this Note has been implemented in LISP for distribution with the ROSIE programming language, which has a highly ambiguous English-like syntax. The Note describes an approach to disambiguation and code generation, as well as an algorithm for constructing the extended LR parse table.

A RAND NOTE

N-2841-DARPA

A Table–Driven Approach to Fast Context–Free Parsing

James R. Kipps

December 1988

Prepared for
The Defense Advanced Research Projects Agency

PREFACE

The work reported here began in connection with the ROSIE[1] Language Development Project, which was funded by the Information Sciences and Technologies Office of the Defense Advanced Research Projects Agency under the auspices of The RAND Corporation's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Since the completion of the ROSIE project, this work was continued independently by the author. The intention in publishing this Note, as well as its companion paper *Analysis of a Table-Driven Algorithm for Fast Context-Free Parsing* (P-7452), is to make the parsing techniques applied in ROSIE available for use by others at RAND and elsewhere.

[1] ROSIE is a registered trademark of The RAND Corporation.

SUMMARY

This Note describes a parsing system developed for the ROSIE programming language and loosely based upon Tomita's algorithm for general context-free parsing. Tomita's algorithm is unique in that it defines a bottom-up parser that uses an extended LR parse table. While this algorithm does no better than the theoretical $O(n^3)$ time bound of other general context-free parsing algorithms (where n is the length or the sentence being parsed), its use of parse tables eliminates an $O(n^2)$ component common to these algorithms. This makes Tomita's algorithm efficient enough for practical application.

The parsing system described here has been implemented in LISP for distribution with the ROSIE programming language, which has a highly ambiguous English-like syntax. An approach to disambiguation and code generation is also described, as is an algorithm for constructing the extended LR parse table.

ACKNOWLEDGMENTS

I would like to thank Ed Hall for reviewing this Note and wading through my sometimes abstruse notational conventions. His comments contributed to several improvements in the clarity of presentation. I would also like to thank Judith Westbury for editing this Note and Mary Aguilar for her assistance in processing it.

CONTENTS

FIGURES

I. INTRODUCTION

A common task for large software systems is the translation of textual input into representations that can be manipulated programmatically. Typically, this task is handled by a separate "translator" subsystem, such as the compiler or interpreter of a programming language, or an interactive human interface or user front-end. The central component of a translator system is a parser, which recognizes whether the input text is syntactically correct and outputs a tree denoting its grammatical structure. While there are many parsing algorithms, only a few are efficient enough to have any practical utility for difficult translation tasks. Available software tools for automating the translation process are limited to these algorithms and the restricted set of languages they recognize. While this set includes most conventional programming languages, it excludes languages that begin to approach natural language in terms of expressiveness, clarity, and readability. The example that sparked this work is ROSIE (Kipps et al., 1987), a language for applications in artificial intelligence (AI) with a highly ambiguous, English-like syntax.

During the development of ROSIE we experimented with several parsing techniques. One was based on a multi-track algorithm (Irons, 1971), and another was based on an unpublished technique developed by Ross Quinlan. Later, these approaches were discarded in favor of a new technique that eventually evolved into the parsing system described here. A similar technique was developed independently at Carnegie-Mellon University (Tomita, 1985a,b). Tomita defines his algorithm as a variation on more conventional parsing technology, with the result that it is easier to understand and explain. We have since refined the design of ROSIE's parsing system after the pattern of Tomita's algorithm.

A detailed presentation of this parsing system is provided in the remainder of this Note. Background information is provided in the next three sections: An overview of parsing concepts is presented in Section II; terminology is defined in Section III; and standard LR parsing technology and its limitations are reviewed in Section IV. An informal outline of Tomita's algorithm is presented in Section V, with an annotated example appearing in Section VI. The algorithm is described formally as a recognizer in Section VII, while it is described as a parser that returns all derivation trees in Section VIII. A discussion of ambiguity and its resolution appears in Section IX, including techniques used in ROSIE to select the best derivation tree. The approach used by ROSIE for code generation given a derivation tree is outlined in Appendix A, and an algorithm for constructing the extended LR parse table used by the parser is presented in Appendix B.

II. BACKGROUND

Parsing techniques have been actively studied since the early days of computer science. This research led to the development of *context-free grammars* (CFGs) (Chomsky, 1956), which have been used extensively for describing the syntax of programming and natural languages.[1] Numerous algorithms have been developed to recognize sentences in languages so described. Some of these algorithms are general, in the sense that they are applicable to all or most CFGs; others are more restricted and are applicable to only a small subclass of CFGs (including the grammars of most programming languages). These latter algorithms, e.g., the LL, operator precedence, predictive, and LR parsing algorithms (Aho and Ullman, 1972), are typically more efficient than the former because they take advantage of inherent features in the class of grammars they recognize. This efficiency makes them appropriate for use in realistic applications; translators based upon these algorithms are often referred to as *practical parsers.*

Most practical parsers analyze the syntax of their input in a single deterministic pass, without need of backup. Each symbol is examined only once, and, at the time it is examined, there is sufficient information available to make any necessary parsing decisions. In his famous paper, Knuth (1965) established a family of CFGs known as LR(k) grammars and provided an effective test to determine, for a given positive integer k, whether a grammar belonged to the LR(k) class. The connection to practical parsers mentioned above is that an LR(k) grammar describes a language, all of whose sentences can be parsed in a single backup-free parse, if at most k symbols of look-ahead are available. While LR(k) grammars have a wide coverage, there are still many interesting languages, such as ROSIE, for which no LR(k) grammar exists.

The primary constraint imposed by the restricted algorithms used in practical parsers involves the level of *ambiguity* they allow. A language is ambiguous if, for some input text, more than one parse tree can be generated. If *disambiguation rules* exist for selecting the "right" parse tree, then the ambiguity is *resolvable*. Resolvable ambiguity is an important aspect of a language. The grammar of a language gives it syntactic structure; ambiguity relaxes the rigidity of this structure. Examples of resolvable ambiguities found in conventional programming languages include arithmetic expressions and if-then-else statements:

[1] Chomsky only applied CFGs to natural language. A similar notation for describing programming languages was developed independently (Backus, 1959).

examples found in natural language include the attachment of prepositions and relative clauses. An ambiguous language trades syntactic precision for conciseness, informality, and readability. However, when the ambiguities are resolvable, then the language regains its precision (at least at a mechanical level). When the disambiguation rules agree with the intuitive expectations of users, then precision is also regained at the human level. Although some practical parsers, such as those produced by YACC (Johnson, 1975), permit a small degree of ambiguity in the languages they recognize, there are nonetheless ambiguous languages, like ROSIE, which have reasonable and understandable disambiguation rules but which cannot be recognized by these parsers. Such languages require the power of general context-free parsing.

The general context-free parsing algorithms, e.g., Earley's algorithm (Earley, 1968; 1970) and the Cocke-Younger-Kasami algorithm (Younger, 1967), are necessarily less efficient than the restrictive algorithms because they must simulate a multiple-path, nondeterministic pass over the input tokens (as opposed to a single-path, deterministic pass). While many of the general algorithms can be shown to theoretically run as efficiently as the restricted algorithms on a large subclass of CFGs, in terms of actual run-time performance they are still too slow to be considered practical. Recently, however, Tomita (1985a,b) introduced a general context-free parsing algorithm that he defines as a variation on standard LR(k) parsing, i.e., a table-driven, bottom-up parsing algorithm. The benefit of this approach is that it eliminates the need to expand alternatives of a nonterminal at parse time (i.e., what Earley calls the *predictor* operation). For Earley's algorithm, the predictor operation is one of two $O(n^2)$ components. While eliminating this operation will not change the upper-time bound of $O(n^3)$, it can make the difference between a practical parser and an interesting toy.

Although Tomita's algorithm has been shown to belong to the same complexity class as Earley's and other general algorithms (Kipps, 1988), in realistic applications it is unlikely to ever realize worse than $n \log n$ time. This is because useful languages and sentences normally must be human-readable as well as machine-readable. This places a natural limit on the complexity of the input a parser might expect to receive. Because Tomita's algorithm takes advantage of parse tables similar to those found in practical parsers, implementations of this algorithm have the potential for run-time performance that is comparable to the restricted algorithms, while retaining the power to recognize a far wider range of languages.

III. TERMINOLOGY

A *language* is a set of strings over a finite set of symbols. These symbols are called *terminals* and are denoted by lowercase letters, e.g., a, b, c. A *context-free grammar* is used as a formal device for specifying which strings are in a language; hereafter, grammar is used to mean context-free grammar. Besides terminals, a grammar uses two other sets of symbols called *preterminals* and *nonterminals*. Preterminals define the lexical classes of a language, while nonterminals define its syntactic classes; preterminals are denoted by lowercase letters preceded by an asterisk, e.g., *a, *b, *c, and nonterminals by capital letters, e.g., A, B, C. Together the terminals (T), preterminals (P), and nonterminals (N) of a language make up its *vocabulary* (V). Individual vocabulary symbols are denoted by capital letters in italics, e.g., A, B, C, while strings of zero or more vocabulary symbols are denoted by Greek letters, e.g., α, β, γ. The empty string is ϵ.

A grammar consists of a finite set of rewrite rules or *productions* of the form

$$\mathtt{A} \rightarrow \alpha$$

where the A component is called the *left-hand side* of the production, and the α component is called its *right-hand side*. The nonterminal that stands for "sentence" is called the *root* (R) of the grammar. Productions with the same nonterminal on their left-hand side are called *alternatives* of that nonterminal. For example,

$$\mathtt{A} \rightarrow \alpha$$
$$\mathtt{A} \rightarrow \beta$$
$$\mathtt{A} \rightarrow \gamma$$

are alternatives of A. Productions of the form

$$\mathtt{A} \rightarrow \epsilon$$

are called *null productions*.

The rest of the definitions are given with respect to a particular grammar G, called the *source grammar*. We write

$$\alpha \Rightarrow \beta$$

if $\exists \gamma, \delta, \eta$,A such that $\alpha = \gamma \mathtt{A} \delta$ and $\beta = \gamma \eta \delta$ and $\mathtt{A} \rightarrow \eta$ is a production. We write

$$\alpha \stackrel{*}{\Rightarrow} \beta$$

(α *derives* β) if $\exists \alpha_0, \alpha_1, \cdots, \alpha_m$ ($m \geq 0$) such that

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_m = \beta.$$

The sequence $\alpha_0, \cdots, \alpha_m$ is called a derivation (of β from α).

A *sentential form* is a string α such that the root $R \stackrel{*}{\Rightarrow} \alpha$. A *sentence* is a sentential form consisting entirely of terminal symbols. The *language defined by a grammar, L(G),* is the set of G's sentences. Any sentential form may be represented in at least one way as a *derivation tree*, reflecting the steps made in deriving it (though not the order of the steps). The *degree of ambiguity* of a sentence is the number of its distinct derivation trees. A sentence is *unambiguous* if it has degree 1 of ambiguity. A grammar is *unambiguous* if each of its sentences is unambiguous.

A *recognizer* is an algorithm that takes as its input a string and either *accepts* or *rejects* it, depending on whether the string is a sentence of the language defined by the source grammar. A *parser* is a recognizer that outputs the set of all legal derivation trees of a string upon acceptance.

IV. REVIEW OF LR PARSING

Later sections of this Note assume familiarity with standard LR parsing, the exact definition and operation of which can be found in Aho and Ullman (1972). In this section, we will briefly review these concepts by examining a generic LR parser as a recognizer. This discussion will also highlight the inherent limitations of standard LR parsing.

LR parsers derive their name from the fact that they scan their input from *L*eft-to-right and construct a *R*ightmost derivation tree in reverse. For all but trivial examples, LR parsers are too complex to implement by hand. Their implementation is straightforward to automate, giving rise to a class of software tools called LR parser generators. Such tools take a context-free grammar as input and output an LR parser for the language described.

LR parsers can be thought of as consisting of two parts: a *driver* routine, and a *parse table*. Typically, the parser produced by an LR parser generator will always contain the same driver routine; only the parse table changes. In this way, LR parser generators are actually parse table generators. (A discussion of parse table construction is not important to understanding the operation of the driver routine, which is the focus of this section. Parse table construction is discussed later in Appendix B.)

Figure 4.1 depicts a generic LR parser. It has a input string, a stack, and a parse table. The input string is scanned from left-to-right, one token at a time. Each stack element records a symbol s, called a *parse state* (or *state*).[1] The mth element, labeled s_m, is at the top-of-stack and is referred to as the state of the parser.

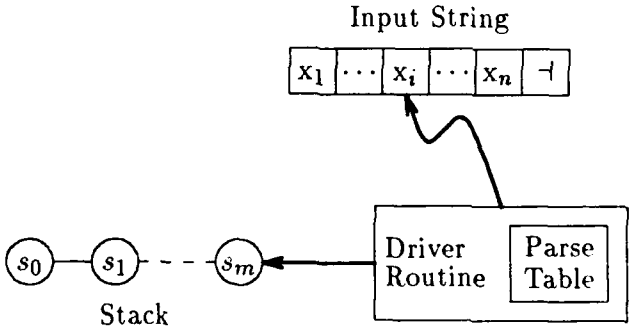


Fig. 4.1—LR parser

[1] Stack elements also record useful information, such as the sequence of derivations made in reaching a parse state. Since we are only considering an LR parser as a recognizer, this other information can be safely ignored.

The parse table consists of two parts: an *action table* and a *goto table*, which are implemented as the look-up functions ACTION and GOTO. The action table maps states and tokens[2] to parse actions. ACTION(s,x) takes a state s and input token x and returns one of four actions:

1. Shift to State s'.
2. Reduce using Production p.
3. Accept.
4. Error.

The goto table maps states and nonterminal symbols to states. GOTO(s,D) takes a state s and nonterminal symbol D and returns a state s', which corresponds to the action 'Shift to State s'.'

Each move of the driver routine is determined by consulting the action table entry for s_m (the current state, i.e., on the top of the stack) and x_i (the current input token). The moves resulting from the four types of actions are as follows:

1. If ACTION(s_m,x_i) = 'Shift to State s',' the driver executes a shift move by creating a new top-of-stack element for s'. Afterwards, input scan is advanced to x_{i+1}, making it the new current input token.
2. If ACTION(s_m,x_i) = 'Reduce using Production p,' the process executes a reduce move. A reduce move corresponds to the recognition of a rightmost derivation of Production p. If Production p has the form
$$D_p \rightarrow C_{p1} \cdots C_{p\bar{p}}$$
where $\bar{p}$ is the length of its right-hand side, then the top $\bar{p}$ stack elements, $s_{m-(\bar{p}-1)} \cdots s_m$, correspond to $C_{p1} \cdots C_{p\bar{p}}$. The process first removes these $\bar{p}$ elements from the stack, making $s_{m-\bar{p}}$ the new top-of-stack. Having recognized the nonterminal D_p, it then executes a shift move to the state s' = GOTO($s_{m-\bar{p}}$,D_p) by creating a new top-of-stack element for s'. Input scan is *not* advanced and the current input token is unchanged.
3. If ACTION(s_m,x_i) = 'Accept,' the driver halts and returns success.
4. If ACTION(s_m,x_i) = 'Error,' the driver executes an error recovery routine, e.g., displaying a message indicating the point in the input string where the error was detected.

The driver operates by first initializing the stack with the *start state* of the parse table (designated State 0) and input scan to the first token of the input string, x_1. It then

[2] For this discussion, we refer to both preterminals and terminals as tokens.

repeatedly executes the moves determined by the action table entry for its state and the current input token until the input string is accepted or rejected.

As an example, consider the simple English-like grammar shown in Figure 4.2. The nonterminals `S`, `NP`, `VP`, and `PP` correspond to four syntactic classes: sentence, noun phrase, verb phrase, and prepositional phrase; the preterminals `*det`, `*n`, `*v`, and `*prep` correspond to four lexical classes: determiner, noun, verb, and preposition. Each sentence in this language contains a noun phrase, followed by a verb phrase, followed by an arbitrary number of prepositional phrases.

The parse table for the example grammar is shown in Figure 4.3. Entries in the action table have the meaning:

- **sh***s* — Shift to State *s*.
- **re***p* — Reduce using Production *p*.
- **acc** — Accept.
- *blank* — Error.

Entries *s* in the goto table mean 'Shift to State *s*.'

(1) `S → NP VP`
(2) `S → S PP`
(3) `NP → *n`
(4) `NP → *det *n`
(5) `PP → *prep NP`
(6) `VP → *v NP`

Fig. 4.2—Example LR grammar

State	Action Table					Goto Table			
	*det	*n	*v	*prep	⊣	NP	PP	VP	S
0	sh3	sh5				8			1
1				sh2	acc		7		
2	sh3	sh5				6			
3		sh4							
4			re4	re4	re4				
5			re3	re3	re3				
6				re5	re5				
7				re2	re2				
8			sh9					11	
9	sh3	sh5				10			
10				re6	re6				
11				re1	re1				

Fig. 4.3—LR parse table

Next consider the moves made by the driver routine in parsing the input 'I saw the man'; assume that I and man are equivalent to *n, saw to *v, and the to *det. Figure 4.4 traces the nine moves made by the driver. (The n + 1st input token '⊣' denotes end-of-sentence.)

The column labeled 'Stack' shows the contents of the stack (states) at the beginning of each move; top-of-stack is the rightmost state. Under the column labeled 'Input,' a dot 'o' prefixes the current input token. For illustrative purposes, when a segment of the input has been recognized in a derivation of some nonterminal (i.e., after a reduce move), it is replaced by that nonterminal. The column labeled 'Action' shows the action table entry for each move; for reduce moves, the subsequent goto table entry is also shown.

In [1], the parser is in State 0 scanning the first input token I. ACTION(0,*n) = sh5, so the driver executes a shift move, making State 5 the current state, and input scan advances to saw. In [2], the driver is in State 5 scanning saw. ACTION(5,*v) = re3, so the driver executes a reduce move. The length of the right-hand side of Production 3, NP → *n, equals one, so one element is removed from the stack, making State 0 the current state. The parser then consults the goto table entry for State 0 and NP, the left-hand side of the production. GOTO(0,NP) = 8, so the driver executes a shift move to State 8. In [3], the current state is State 8 and I has been replaced by NP, but the input scan is still at saw. Moves [3] through [9] are determined in a manner similar to [1] and [2].

The principal limitation of LR parsers comes from the fact that they only recognize a restricted set of context-free grammars. The general restriction is that the grammars must be unambiguous (or unambiguous with k-symbol look-ahead).[3] Most conventional programming languages can be described by an LR(k) grammar. However, context-free

	Stack	Input	Action	
[1]	0	oI saw the man ⊣	sh5	
[2]	0 5	NP osaw the man ⊣	re3	GOTO(0,NP) = 8
[3]	0 8	NP osaw the man ⊣	sh9	
[4]	0 8 9	NP saw othe man ⊣	sh3	
[5]	0 8 9 3	NP saw the oman ⊣	sh4	
[6]	0 8 9 3 4	NP saw the man o⊣	re4	GOTO(9,NP) = 10
[7]	0 8 9 10	NP saw NP o⊣	re6	GOTO(8,VP) = 11
[8]	0 8 11	NP VP o⊣	re1	GOTO(0,S) = 1
[9]	0 1	S o⊣	acc	

Fig. 4.4—Moves of an LR parser

[3] Although some LR parser generators allow a small degree of ambiguity to exist.

languages are inherently ambiguous (Ginsburg and Ullian, 1966), which means that there will be interesting and desirable languages for which no LR(k) grammar exists for any k.

To illustrate, suppose we wished to enhance the grammar in Figure 4.2 to allow a noun phrase to consist of an arbitrary number of prepositional phrases. We can do this by adding the production

$$\texttt{NP} \rightarrow \texttt{NP PP}$$

As we will see in Section 5, this simple change makes the grammar non-LR(k). What this means to our example is that the action table will contain two double-action entries, denoting the two uses of a prepositional phrase. Since the driver is a deterministic process with only a single stack, it can follow one of these actions but not both. To follow both, the drive must become nondeterministic, using some form of search to parse the input. In fact, this is the essence of Tomita's algorithm, as the remainder of this Note explains.

V. TOMITA'S ALGORITHM

The following is an informal description of Tomita's algorithm as a recognizer. (Familiarity with standard LR parsing is assumed.) Tomita views his algorithm as a variation on standard LR parsing. The algorithm takes a shift-reduce approach, using an extended LR parse table to guide its actions. The change the algorithm makes to the parse table is to allow it to contain multiple actions per entry, i.e., at most one *shift* action or *accept* action and any number of *reduce* actions. Thus, the parse table can no longer be used for strictly deterministic parsing; some search must be done. The algorithm emulates a nondeterministic parse with pseudo-parallelism. It scans an input string $x_1 \cdots x_n$ from left to right, following all paths in a breath-first manner and merging like subpaths when possible to avoid redundant computations.

An example non-LR grammar is shown in Figure 5.1. This is a simplistic subset of English, where the nonterminals `S`, `NP`, `VP`, and `PP` define the syntactic classes sentence, noun phrase, verb phrase, and prepositional phrase; and the preterminals `*det`, `*n`, `*v`, and `*prep` define the lexical classes determiner, noun, verb, and preposition. The parse table for this grammar is shown in Figure 5.2.

In building the parser table the grammar is augmented by a 0th production

$$D_0 \rightarrow R \dashv$$

where R is the root of the grammar and where the symbol '$\dashv$' is a special terminal that denotes end-of-sentence and appears only as the last symbol of an input string. Entries **sh***s* in the action table indicate the action 'Shift to State *s*,' and entries **re***p* indicate the action 'Reduce constituents on the stack according to Production *p*.' The entry **acc** indicates 'Accept,' and blanks indicate 'Error.' Entries *s* in the goto table indicate the action 'After a reduce action, shift to State *s*.' In Figure 5.2, there are two multi-action entries in States 3 and 11 under the column labeled `*prep`.

```
(1)   S → NP VP
(2)   S → S PP
(3)  NP → *n
(4)  NP → *det *n
(5)  NP → NP PP
(6)  PP → *prep NP
(7)  VP → *v NP
```

Fig. 5.1—Example non-LR grammar

State	Action Table					Goto Table			
	*det	*n	*v	*prep	⊣	NP	PP	VP	S
0	sh5	sh7				9			1
1				sh2	acc		8		
2	sh5	sh7				3			
3			re6	re6,sh2	re6		4		
4			re5	re5	re5				
5		sh6							
6			re4	re4	re4				
7			re3	re3	re3				
8				re2	re2				
9			sh10	sh2			4	12	
10	sh5	sh7				11			
11				re7,sh2	re7		4		
12				re1	re1				

Fig. 5.2—LR parse table with multiple entries

The algorithm operates by maintaining a number of parsing *processes* in parallel. Each process has a stack, scans the input string from left-to-right, and, thus, behaves basically the same as the single parsing process in standard LR parsing. Each stack element is labeled with a parse state and points to its parent, i.e., the previous element on a process's stack. We call the top-of-stack the *current state* of a process.

Actually, each process does not maintain its own separate stack. Rather, these "multiple" stacks are represented using a single directed acyclic (but reentrant) graph called a *graph-structured stack*, an example of which is shown in Figure 5.3.[1]

Each stack element corresponds to a vertex of the graph. (In Figure 5.3, circles represent the vertices of the graph, where each circle is labeled with a parse state.) Each leaf of the

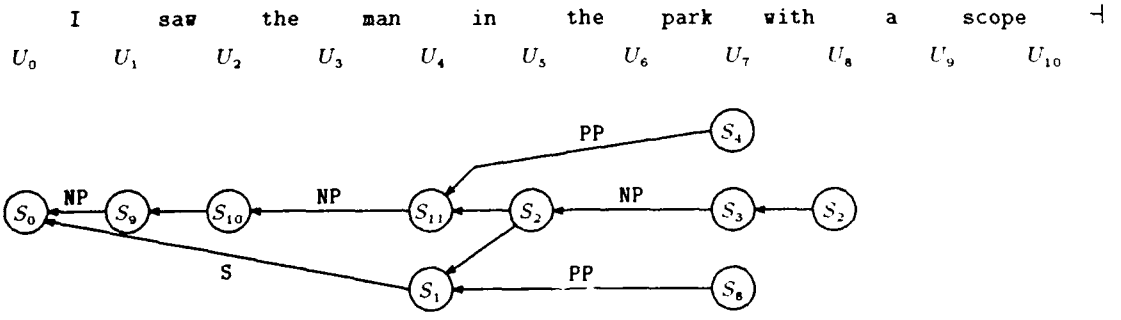


Fig. 5.3—Example graph-structured stack

[1] Figure adapted from examples given in Section VI.

graph acts as a distinct top-of-stack to a process. (There are three processes depicted in Figure 5.3: one in State 4, one in State 2, and one in State 8.) The root of the graph acts as a common bottom-of-stack. The edge between a vertex and its parent is directed toward the parent. (As a convenience to the reader, edges in Figure 5.3 are labeled when they denote the recognition of a nonterminal.) Because of the reentrant nature of the graph (as explained below), a vertex may have more than one parent. (In Figure 5.3, the vertex under U_5 has two parent vertices.)

The leaves of the graph grow in stages. Each stage U_i corresponds to the ith symbol x_i from the input string. After x_i is scanned, the leaves in stage U_i are in a one-to-one correspondence with the algorithm's *active* processes, where each process references a distinct leaf of the graph and treats that leaf as its current state. Upon scanning x_{i+1}, an active process can either (1) add an additional leaf to U_i, or (2) add a leaf to U_{i+1}. Only processes that have added leaves to U_{i+1} will be active when x_{i+2} is scanned. (In Figure 5.3, x_i = `with`; two leaves have been added to U_i and one to U_{i+1}.)

In general, a process behaves in the following manner. On x_i, each active process (corresponding to the leaves in U_{i-1}) executes the entries in the action table for x_i given its current state. When a process encounters multiple actions, it *splits* into several processes (one for each action), each sharing a common top-of-stack. When a process encounters an error entry, the process is discarded (i.e., its top-of-stack vertex sprouts no leaves into U_i by way of that process). All processes are synchronized, scanning the same symbol at the same time. After a process shifts on x_i into U_i, it waits until there are no other processes that can act on x_i before scanning x_{i+1}.

The Shift Action. A process (with top-of-stack vertex v) shifts on x_i from its current state s to some successor state s' by

(1) creating a new leaf v' in U_i labeled s';

(2) placing an edge from v' to its top-of-stack v (directed towards v); and

(3) making v' its new top-of-stack vertex (in this way changing its current state).

Any successive process shifting to the same state s' in U_i is *merged* with the existing process to form a single process whose top-of-stack vertex has multiple parents, i.e., by placing an additional edge from the top-of-stack vertex of the existing process in U_i to the top-of-stack vertex of the shifting process. The merge is done because, individually, these processes would behave in exactly the same manner until a reduce action removed the vertices labeled s' from their stacks. Thus, merging avoids redundant computation. Merging also ensures that each leaf in any U_i will be labeled with a distinct parse state, which puts a finite

upper-bound on the possible number of active processes and, thus, limits the size of the graph-structured stack.

The Reduce Action. A process executes a reduce action on a production p by following the chain of parent links down from its top-of-stack vertex v to the ancestor vertex from which the process began scanning for p earlier, essentially "popping" intervening vertices off its stack. Since merging means a vertex can have multiple parents, the reduce operation can lead back to multiple ancestors. When this happens, the process is again split into separate processes (one for each ancestor). The ancestors will correspond to the set of vertices at a distance $\bar{p}$ from v, where $\bar{p}$ equals the number of symbols in the right-hand side of the pth production. Once reduced to an ancestor, a process shifts to the state s' indicated in the goto table for D_p (the nonterminal on the left-hand side of the pth production) given the ancestor's state. A process shifts on a nonterminal much as it does a terminal, with the exception that the new leaf is added to U_{i-1} rather than U_i. (A process can only enter U_i by shifting on x_i.)

The algorithm begins with a single initial process whose top-of-stack vertex is the root of the graph-structured stack. It then follows the general procedure outlined above for each symbol in the input string, continuing until there are either no leaves added to U_i (i.e., *no more active processes*), *which denotes rejection*, or a process executes the accept action on scanning the $n + 1$st input symbol '$\dashv$,' which denotes *acceptance*. (Figure 5.3 shows an instance of a graph-structured stack in which three vertices were added to U_7 as a result of reduce actions while scanning `with`; a vertex was added to U_8 as a result of a shift action on `with` in State 3.)

VI. EXAMPLE RECOGNITION

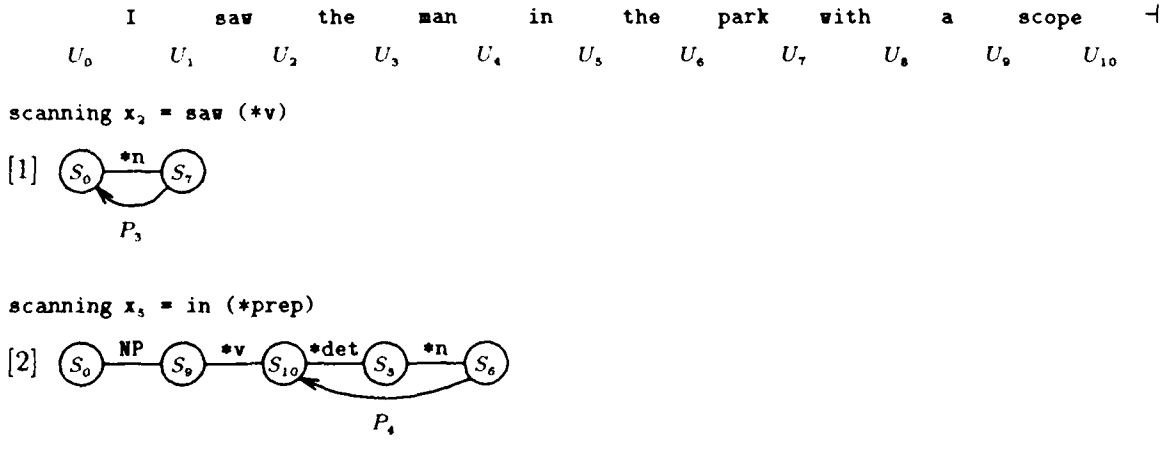
To illustrate the mechanics of Tomita's algorithm as a recognizer, consider the following example, which uses the grammar in Figure 5.1 and parse table in Figure 5.2 to analyze the sentence

'I saw the man in the park with a scope ⊣'

Assume that the tokens I, man, park, and scope belong to the lexical class *n, that saw belongs to *v, that in and with belong to *prep, and that a and the belong to *det.

The diagrams shown below trace the growth of the graph-structured stack from U_0 through U_{10}. Each circle is a vertex of the graph labeled by a parse state. The line segments connecting vertices are edges created by shift actions (i.e., from the state of the left vertex, on the grammar symbol labeling the segment, and into the state of the right vertex). Backward arrows indicate a reduce action back to the vertex at the head of the arrow. Each arrow is labeled P_i, indicating that an instance of the ith production was recognized. After reducing back to the earlier vertex, a shift is done on nonterminal D_i (the left-hand side of the ith production).

In the beginning, there is only a single active process, denoted by a single vertex in U_0 labeled S_0. The only action from State 0 on I is sh7, so in [1] the process creates a new vertex labeled S_7, adding it to U_1. The only action in State 7 on saw is re3, so the process reduces back one vertex (the length of Production 3) as indicated by the backward arrow. From State 0 [2], the process shifts to State 9 on NP, State 10 on saw, State 5 on the, and State 6 on man. From State 6, it reduces back two vertices using Production 4.

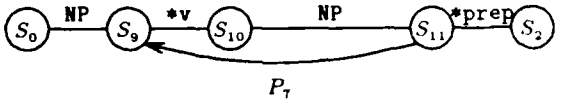


I saw the man in the park with a scope ⊣

U_0 U_1 U_2 U_3 U_4 U_5 U_6 U_7 U_8 U_9 U_{10}

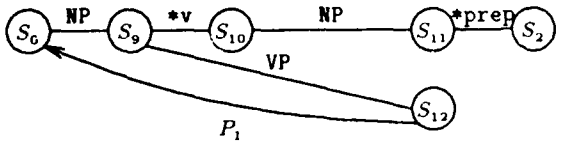
scanning x_6 = the (*det)

[3]



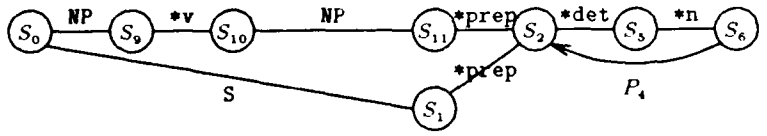
scanning x_6 = the (*det)

[4]



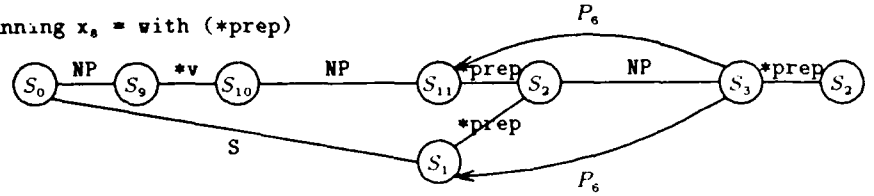
scanning x_8 = with (*prep)

[5]



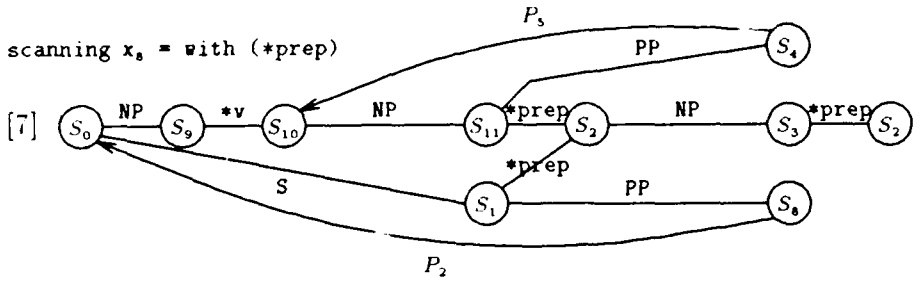
scanning x_8 = with (*prep)

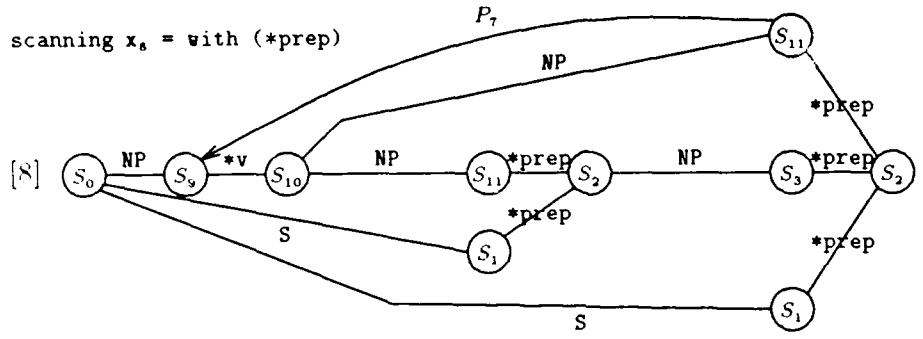
[6]

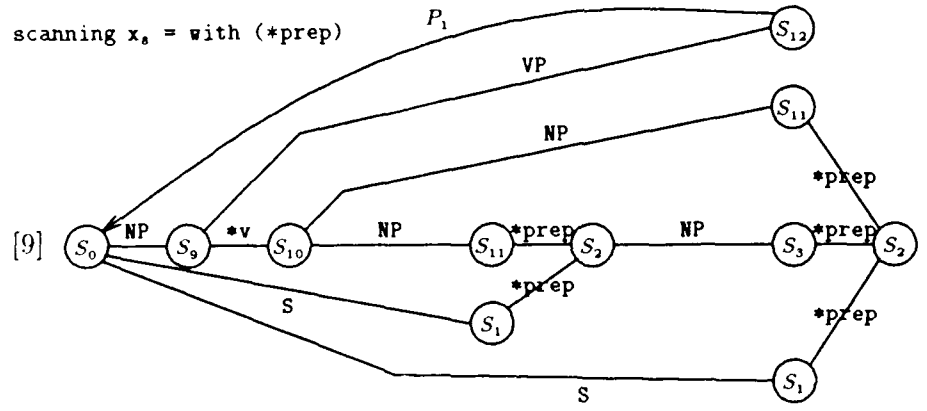


From State 10 [3], the process shifts to State 11 on NP, where it finds two actions on in, sh2 and re7. The process splits in two. In [4], one process follows the shift action, while the second follows the reduce action. In [5], the second process has shifted on in, entering the same state as the first and merging with it into a single process again. This merged process shifts first on the and then park to State 6, where it reduces on with using Production 4. In [6], the process encounters another two-action state. It splits in two: one process follows the shift action; the other follows the reduce action. As the latter process goes back past the earlier merged vertex, it splits once more.

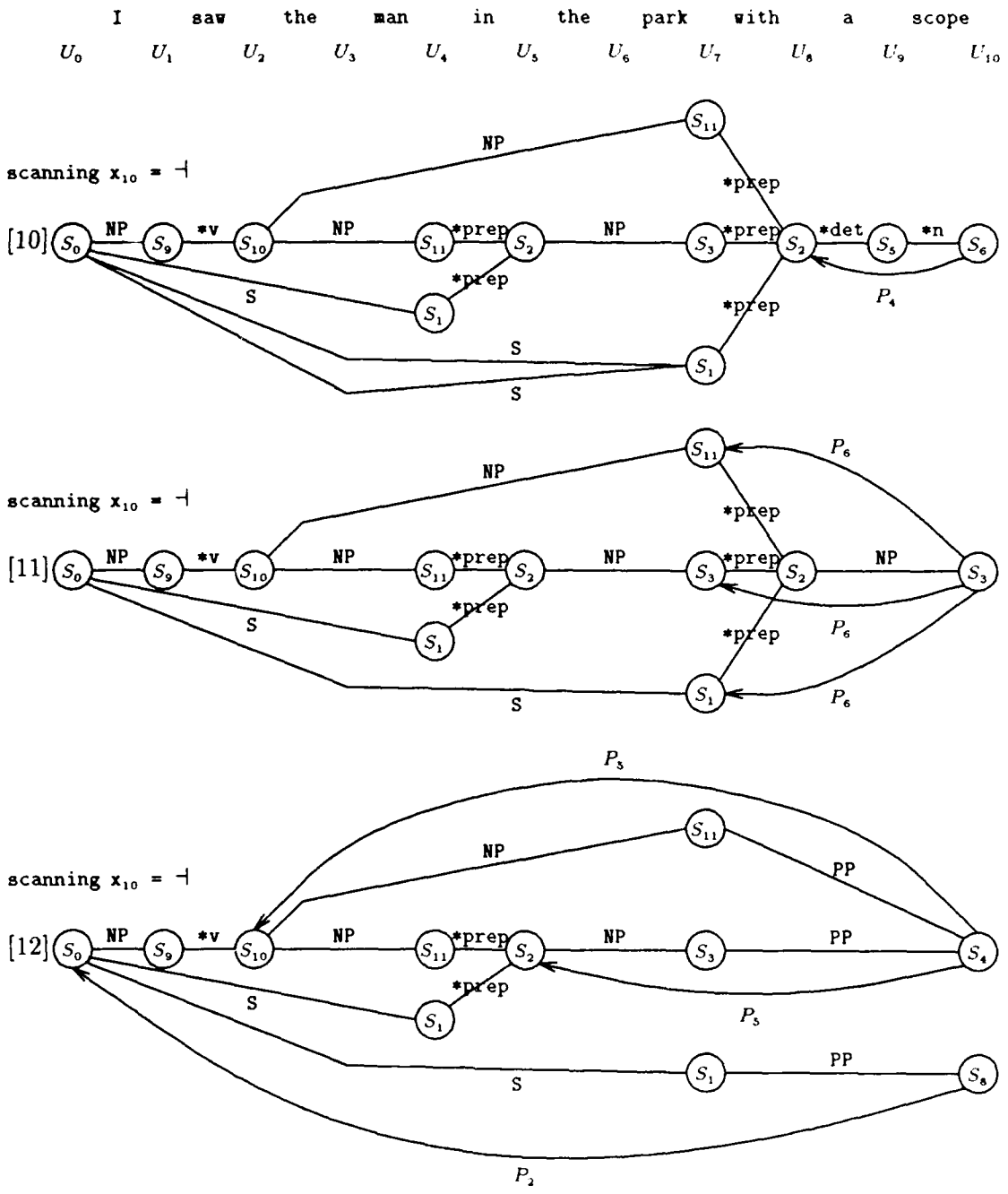
I saw the man in the park with a scope ⊣
U_0 U_1 U_2 U_3 U_4 U_5 U_6 U_7 U_8 U_9 U_{10}



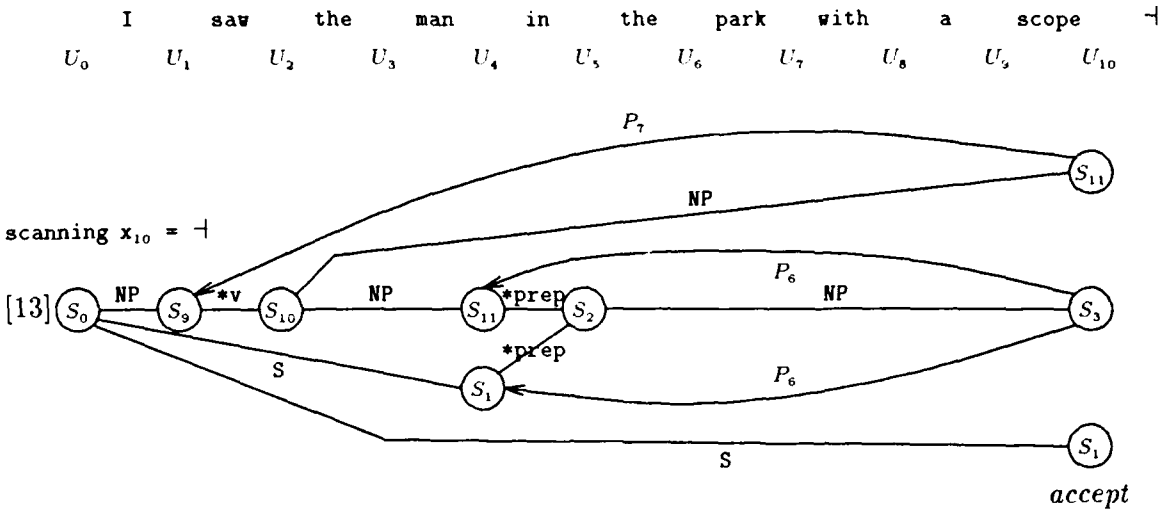




In [7], these processes reduce again. Finally in [8], they shift on **with** to merge with the process that originally followed the shift action earlier in [4]. Note that the process in State 11 also has reduce action on **with**. It splits and reduces twice: first (in [8]) using Production 7 and then (in [9]) using Production 1.



Eventually in [10], this process shifts to State 1 on S. Because another process has already made the same shift, the recognizer detects an ambiguity and discards the second process. Meanwhile, the three processes that merged on **with** at State 2, shift on **the** and **scope** to State 6, and reduce on '⊣' using Production 4. In [11], the merged processes reduce back past the merging vertex, where they split back into three separate processes, two of which re-merge on PP in [12].



Finally in [13], one of the processes reaches a state with an accept action on the end-of-sentence symbol '⊣,' and the sentence is recognized as belonging to the language.

VII. THE RECOGNIZER

In this section, a more precise definition of the algorithm from Section V is presented as a recognizer for an input string $x_1 \cdots x_n$. This definition is understood to be with respect to an extended LR parse table (with start state S_0) constructed from a source grammar G.

Notation. Number the productions of G arbitrarily $1, \cdots, d$, where each production is of the form

$$D_p \rightarrow C_{p1} \cdots C_{p\bar{p}} \qquad (1 \leq p \leq d)$$

and where $\bar{p}$ is the number of symbols on the right-hand side of the pth production.

Definition. The entries of the extended LR parse table are accessed with the functions ACTIONS and GOTO.

- ACTIONS(s,x) returns a set of actions from the action table along the row of state s under the column labeled x. This set will contain no more than one of a shift action shs' *or* an accept action acc; it may contain any number of reduce actions re$p$. (If the action set is empty (which corresponds to an error), no actions are taken and the process never advances.)
- GOTO(s,D_p) returns a state s' from the goto table along the row of state s under the column labeled with nonterminal D_p.

Definition. Each *vertex* of the graph-structured stack is a triple $\langle i, s, l \rangle$, where i is an integer corresponding to the ith input symbol scanned (i.e., the time at which the vertex was created as a leaf), s is a parse state (corresponding to a row of the parse table), and l is a set of parent vertices. The *processes* described in the last section are represented implicitly by the vertices in successive U_i's. The root of the graph-structured stack, and hence the initial process, is the vertex $\langle 0, S_0, \emptyset \rangle$.

The Recognizer. The recognizer is a function of one argument REC($x_1 \cdots x_n$). It calls upon the functions SHIFT(v,s) and REDUCE(v,p). SHIFT(v,s) either (1) adds a new leaf to U_i labeled with parse state s whose parent is vertex v or (2) merges vertex v with the parents of an existing leaf. REDUCE(v,p) executes a reduce action from vertex v using production p. REDUCE calls upon the function ANCESTORS(v,$\bar{p}$), which returns the set of all ancestor vertices a distance of $\bar{p}$ from vertex v. These functions, which vary somewhat from the formal definition given in Tomita (1985a),[1] are defined in Figure 7.1.

---

[1] The changes to Tomita's algorithm make it easier to describe but do not alter it significantly. In particular, Tomita's functions REDUCE and REDUCE-E have been collapsed into one function.

```
    REC(x_1 ··· x_n)
[1]   let x_{n+1} := ⊣
      let U_i := [ ]    (0 ≤ i ≤ n)
[2]   let U_0 := [⟨0, S_0, ∅⟩]
[3]   for i from 1 to n + 1
        let P := [ ]
[4]     for ∀v = ⟨i − 1, s, l⟩ s.t. v ∈ U_{i−1}
          let P := P ∘ [v]
[5]       if ∃'sh s'' ∈ ACTIONS(s,x_i), SHIFT(v,s')
          for ∀'re p' ∈ ACTIONS(s,x_i), REDUCE(v,p)
          if 'acc' ∈ ACTIONS(s,x_i), accept
[6]     if U_i is empty, reject

    SHIFT(v,s)
[7]   if ∃v' = ⟨i, s, l⟩ s.t. v' ∈ U_i
        let l := l ∪ {v}
      else
        let U_i := U_i ∘ [⟨i, s, {v}⟩]

    REDUCE(v,p)
[8]   for ∀v_1' = ⟨j', s', l_1'⟩ s.t. v_1' ∈ ANCESTORS(v, p̄)
        let s'' := GOTO(s', D_p)
[9]     if ∃v'' = ⟨i − 1, s'', l''⟩ s.t. v'' ∈ U_{i−1}
[10]      if v_1' ∈ l''
            do nothing (ambiguous)
          else
[11]      if ∃v_2' = ⟨j', s', l_2'⟩ s.t. v_2' ∈ l''
            let v_c := ⟨i − 1, s'', {v_1'}⟩
            for ∀'re p' ∈ ACTIONS(s'',x_i), REDUCE(v_c,p)
          else
[12]        let l'' := l'' ∪ {v_1'}
[13]        if v'' ∈ P
              let v_c := ⟨i − 1, s'', {v_1'}⟩
              for ∀'re p' ∈ ACTIONS(s'',x_i), REDUCE(v_c,p)
        else
[14]      let U_{i−1} := U_{i−1} ∘ [⟨i − 1, s'', {v_1'}⟩]

    ANCESTORS(v = ⟨j, s, l⟩, c)
[15]  if c = 0
        return({v})
      else
        return(⋃_{v'∈l} ANCESTORS(v', c − 1))
```

Fig. 7.1—The recognizer

In REC, [1] adds the end-of-sentence symbol '⊣' to the end of the input string; [2] initializes the root of the graph-structured stack; [3] iterates through the symbols of the input string. On each symbol x_i, [4] processes the vertices (denoting the active processes) of successive U_{i-1}'s, adding each vertex to P to signify that it has been processed. On each vertex v, [5] executes the shift, reduce, and accept actions from the action table given v's state s. After processing the vertices in U_{i-1}, [6] checks whether a vertex was added to U_i, ensuring that at least one process is still active before scanning x_{i+1}.

In SHIFT, [7] shifts a process into state s by adding a vertex to U_i labeled s. If a vertex labeled s already exists, v is added to its parents, merging processes; otherwise, a new vertex is created with a single parent v.

In REDUCE, [8] iterates through the ancestor vertices a distance of $\bar{p}$ from v, setting s'' to the state indicated in the goto table under D_p given the ancestor's state s'. Each ancestor vertex v_1' is shifted into U_{i-1} on s''. [9] checks whether such a vertex v'' already exists. (If not, [14] adds a vertex labeled s'' to U_{i-1}.) If v'' does already exist, [10] checks that a shift from the current ancestor v_1' has not already been made. (If it has, then some segment of the input string has been recognized as an instance of the same nonterminal D_p in two different ways, and the current derivation can be discarded as ambiguous; otherwise, v_1' is merged with the parents of the existing vertex.) Before merging, [11] checks whether v_1' is a "clone" vertex, created by [13] in an earlier call to REDUCE (as described below). If v_1' is not a clone, [12] adds it to the parents of v'', merging processes. [13] checks if v'' has already been processed. If so, then it missed reductions (if any) through v_1'. To correct this, v'' is "cloned" into v_c (i.e., a variant on v'' with a single parent v_1'), and all reduce actions executed on v'' are now executed on v_c.

Returning to [11], when reducing on null production, ANCESTORS will return a clone vertex as the ancestor of itself. If a variant v_2' of v_1' already exists in the parents of v'', then v_1' is a clone and v_2' is the vertex from which it was cloned. At this point v'' has already been processed, meaning that there could still be reductions that have not gone through the single parent of v_1'. To correct this, v'' is again cloned, and all reduce actions executed on v'' are executed on the new clone v_c.

Finally, in ANCESTORS, [15] recursively descends the chain of parents of vertex v, returning the set of vertices a distance of c from v.

VIII. THE PARSER

In this section, the recognizer is augmented to become a parser, returning the set of all derivation trees for the input string $x_1 \ldots x_n$ upon acceptance. Essentially, two changes must be made to the algorithm. First, it can no longer halt on the first process to execute the accept action; instead, it must continue processing the active processes until there are none left. Second, each process must record derivations as they are recognized, constructing portions of a derivation tree on the fly. When the parser has finished scanning the input string, it returns the set of derivation trees recorded by all accepting processes. Later, a disambiguation routine can be applied to this set to select the "right" tree, which can then be used for code generation.

A process "recognizes a derivation" whenever it executes a reduce action. Reducing on Production p means that the process has recognized a derivation of the nonterminal D_p. In the recognizer, we followed the chain of ancestor vertices to a distance of $\bar{p}$ from the top-of-stack vertex of the reducing process. For the parser, we can view each such chain having the form

$$v_1, \cdots, v_{\bar{p}}, v'$$

where v_1 is the top-of-stack vertex and v' is the ancestor that began scanning for Production p. The vertices $v_1, \cdots, v_{\bar{p}}$ correspond to the symbols $C_{p1} \cdots C_{p\bar{p}}$ in the right-hand side of the pth production, i.e., the symbols derived from D_p. If we assume that each vertex v_i records d_i, the *recognized derivation* of C_{pi}, then when we shift to s' (the state indicated in the goto table for D_p) and add a vertex v'' labeled s' to U_{i-1}, we can record the recognized derivation of D_p by attaching the tuple $\langle p, \alpha \rangle$ (where $\alpha = [d_1, \cdots, d_{\bar{p}}]$) to the link between v'' and its current parent vertex v'. (This tuple indicates that the process has recognized a derivation of D_p using the pth production to derive $d_1 \cdots d_{\bar{p}}$.) Similarly, when a process adds a vertex v to U_i on a shift action, we must record that it has scanned an input symbol corresponding to some C_{pj}, where C_{pj} is either a terminal or preterminal. To indicate this, we attach the symbol scanned to the link between v and its parent vertex.

In the recognizer, when the reduce action causes two or more processes to shift to s' in U_{i-1}, the processes are merged into a single process. However, when two or more processes shift from the same vertex v' to v'', all but the first are discarded as ambiguous. Note that

these processes all recognize a derivation of the same nonterminal D_p over the same input sequence $x_j \cdots x_i$, i.e.,

$$D_p \stackrel{*}{\Rightarrow} x_j \cdots x_i$$

The recognizer can discard all but one of these processes, because only one is needed in recognizing the entire input string. The parser, on the other hand, cannot discard any of these processes, because each recognizes a different derivation of D_p and will lead to distinct parse trees. So, all ambiguous derivations must be recorded. To do this, we treat the recognized derivation of the nonterminal D_p as a set of tuples $\langle p, \alpha \rangle$, each of which indicates a competing derivation of D_p over the same segment of the input string.

The remainder of this section gives a precise definition of the algorithm as a parser for an input string $x_1 \cdots x_n$. This definition is understood to be with respect to an extended LR parse table (with start state S_0) constructed from a source grammar G.

Definition. A *recognized derivation* is a set of tuples $\langle p, \alpha \rangle$ denoting the recognition of a derivation of D_p using the pth production to derive α. α is a linear list of the form $[d_1, \cdots, d_{\bar{p}}]$, where each d_i is either a recognized derivation of C_{pi}, when C_{pi} is a nonterminal, or a symbol scanned from the input string, when C_{pi} is a terminal or preterminal.

Definition. Each *vertex* (v) of the graph-structured stack is a triple $\langle i, s, l \rangle$, where i and s are as before and where l is a set of tuples $\langle v', d \rangle$. If vertex v' reached v on a shift action on x_i, then $d = x_i$. Otherwise, if vertex v' reached v from a reduce action on Production p, then d is a recognized derivation of D_p.

The Parser. The parser is a function of one argument, PARSE($x_1 \cdots x_n$). It finds the recognized derivation d_R of $x_1 \cdots x_n$ from the root R of grammar G, returning d_R to denote the set of all derivation trees for the input sentence. PARSE calls on the functions SHIFT, REDUCE, and ANCESTORS as defined in Figure 8.1.

Figure 8.1 illustrates the necessary modifications made to the functions of the recognizer to turn it into a parser. In PARSE, [1] adds the end-of-sentence symbol '$\dashv$' to the end of the input string; [2] initializes the root of the graph-structured stack; [3] iterates through the symbols of the input string. On each symbol x_i, [4] processes the vertices (denoting the active processes) of successive U_{i-1}'s, adding each vertex to P to signify that it has been processed. On each vertex v, [5] and [6] execute the shift, reduce, and accept actions from the action table given v's state s. [6] implements the accept action by shifting the accepting process into a dummy accept state S_{accept}. (The accept action corresponds to a shift on the end-of-sentence symbol, which can only appear in recognizing the 0th production, $D_0 \rightarrow R \dashv$. The recognized derivation d attached to the link between v and each of its parents v' is a recognized derivation of the root R.) After processing the vertices in U_{i-1},

```
    PARSE(x_1 ··· x_n)
[1]   let x_{n+1} := ⊣
      let U_i := [ ]    (0 ≤ i ≤ n)
[2]   let U_0 := [⟨0, S_0, ∅⟩]
[3]   for i from 1 to n + 1
        let P := [ ]
[4]     for ∀v = ⟨i − 1, s, l⟩ s.t. v ∈ U_{i−1}
          let P := P ∘ [v]
[5]       if ∃'sh s'' ∈ ACTIONS(s, x_i), SHIFT(v, s')
          for ∀'re p' ∈ ACTIONS(s, x_i), REDUCE(v, p)
[6]       if 'acc' ∈ ACTIONS(s, x_i), accept
[7]     if U_i is empty, reject
[8]   let d_R be [ ]
      for ∀v' = ⟨n, s, l'⟩ s.t. [⟨n+1, S_accept, l⟩] = U_{n+1} ∧ ⟨v', ⊣⟩ ∈ l
        let d_R := d_R ∪ ⋃_{⟨v'', d'⟩ ∈ l'} d'
      Return(d_R)

    SHIFT(v, s)
[9]   if ∃v' = ⟨i, s, l⟩ s.t. v' ∈ U_i
        let l := l ∪ {⟨v, x_i⟩}
      else
        let U_i := U_i ∘ [⟨i, s, {⟨v, x_i⟩}⟩]

    REDUCE(v, p)
[10]  for ∀v_1' = ⟨j', s', l_1'⟩ s.t. ⟨v_1', α⟩ ∈ ANCESTORS(v, p̄, [ ])
        let s'' := GOTO(s', D_p)
[11]    if ∃v'' = ⟨i − 1, s'', l''⟩ s.t. v'' ∈ U_{i−1}
[12]      if ∃d s.t. ⟨v_1', d⟩ ∈ l''
            let d := d ∪ {⟨p, α⟩}
          else
[13]      if ∃d, v_2', l_2' s.t. v_2' = ⟨j', s', l_2'⟩ ∧ ⟨v_2', d⟩ ∈ l''
            let v_c := ⟨i − 1, s'', {⟨v_1', d⟩}⟩
            for ∀'re p' ∈ ACTIONS(s'', x_i), REDUCE(v_c, p)
          else
[14]        let l'' := l'' ∪ {⟨v_1', {⟨p, α⟩}⟩}
[15]        if v'' ∈ P
              let v_c := ⟨i − 1, s'', {⟨v_1', {⟨p, α⟩}⟩}⟩
              for ∀'re p' ∈ ACTIONS(s'', x_i), REDUCE(v_c, p)
        else
[16]      let U_{i−1} := U_{i−1} ∘ [⟨i − 1, s'', {⟨v', {⟨p, α⟩}⟩}⟩]

    ANCESTORS(v = ⟨j, s, l⟩, c, α_{c+1,···,p̄})
[17]  if c = 0
        return({⟨v, α_{1,···,p̄}⟩})
      else
        return(⋃_{⟨v', d⟩ ∈ l} ANCESTORS(v', c − 1, [d] ∘ α_{c+1,···,p̄}))
```

Fig. 8.1 – The parser

[7] checks whether a vertex was added to U_i, ensuring that at least one process is still active before scanning x_{i+1}. After scanning the $n+1$ symbol, [8] extracts d_R, the recognized derivations of R, from the single vertex in U_{n+1}, returning d_R as the set of all derivation trees for the input string.

In SHIFT, [9] shifts a process into state s by adding a vertex to U_i labeled s and recording that x_i was scanned in reaching this vertex. If a vertex labeled s already exists, v is added to its parents, merging processes; otherwise, a new vertex is created with a single parent v.

In REDUCE, [10] iterates through the tuples $\langle v', \alpha \rangle$ returned by ANCESTORS. Each v' is an ancestor a distance of $\bar{p}$ from v, and the corresponding α is a list $[d_1, \cdots, d_{\bar{p}}]$ of derivations recorded at intervening vertices. Each ancestor vertex ${v_1}'$ is shifted into U_{i-1} *on* s'', *the state indicated by the goto table given* D_p and the state of the ancestor. [11] checks whether such a vertex v'' already exists; if not, [16] adds a vertex labeled s'' to U_{i-1}. If v'' does already exist, [12] checks that a shift from the current ancestor ${v_1}'$ has not already been made. (If it has, then some segment of the input string has been recognized as an instance of the same nonterminal D_p in two different ways, and the current derivation $\{\langle p, \alpha \rangle\}$ is merged to d, the other recognized derivation of D_p.) If ${v_1}'$ is not among the parents of v'', [13] checks whether it is a "clone" vertex, created by [15] in an earlier call to REDUCE (as described in Section VII). If ${v_1}'$ is a clone, then there could still be reductions that have not gone through its single parent. To correct this, v'' is again cloned, and all reduce actions executed on v'' are executed on v_c. If ${v_1}'$ is not a clone, [14] adds it to the parents of v'', merging processes. [15] checks if v'' has already been processed. If so, then it missed reductions (if any) through ${v_1}'$. To correct this, v'' is "cloned" into v_c, and all reduce actions executed on v'' are now executed on the clone.

Finally, in ANCESTORS, [17] recursively descends the chain of the parents of vertex v, collecting the derivations associated with intervening vertices and returning the set of vertices a distance of c from v and the list α of derivations.

IX. DERIVATION TREES AND DISAMBIGUATION

In Section VI, we traced the growth of the graph-structured stack during the recognition of the sentence '**I saw the man in the park with a scope.**' The level of ambiguity for this sentence is 5 (i.e., it has 5 distinct derivation trees, as shown below in Figure 9.1). Each internal node has the form $D_p(p)$, where D_p is the nonterminal recognized using

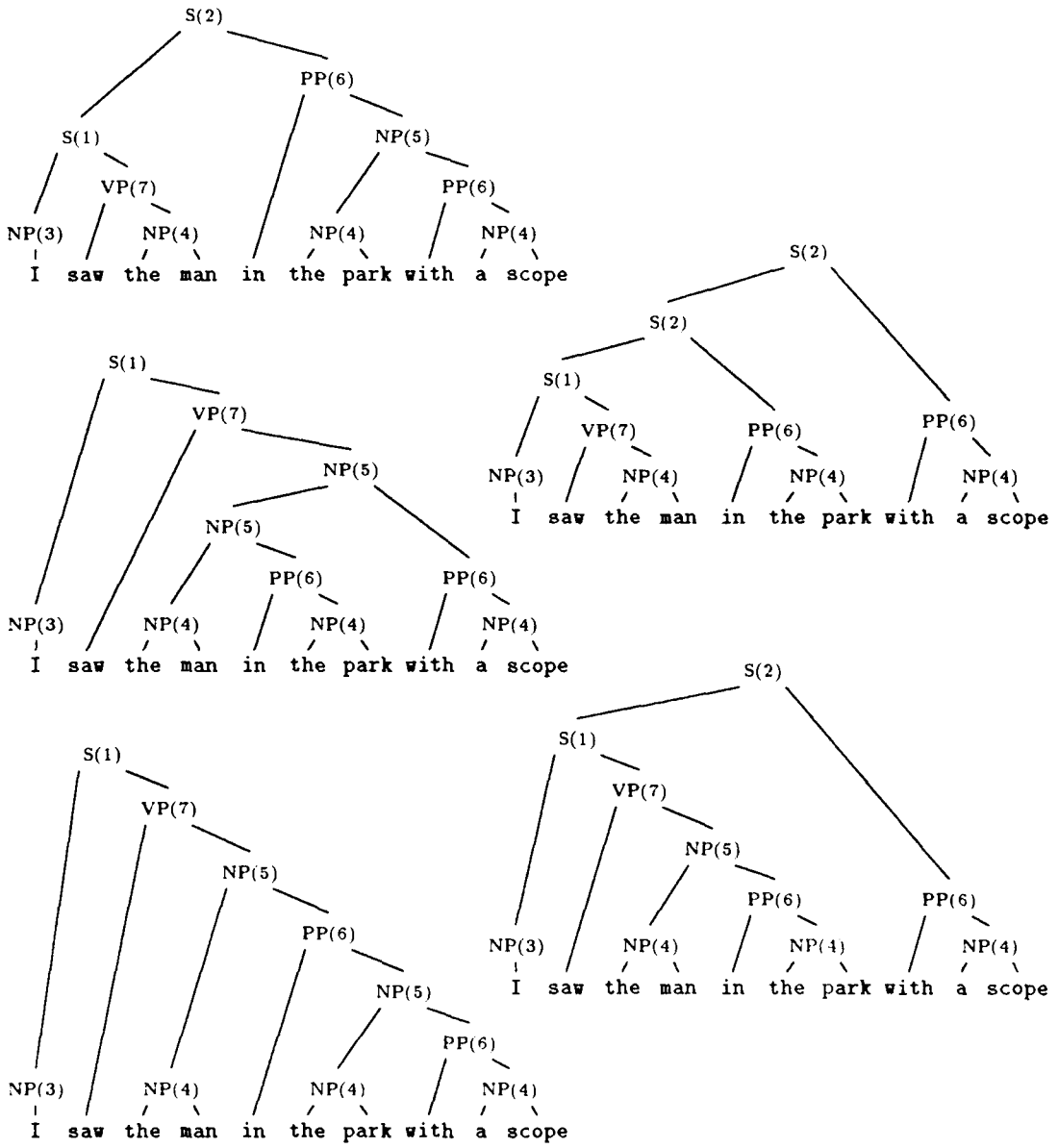


Fig. 9.1—Five derivation trees

Production p. The children of a node, ordered from left to right, correspond to the derivations of $C_{p1}, \cdots, C_{p\bar{p}}$, the symbols appearing in the right-hand side of the pth production.

The parser outputs these trees in a single, compact representation called a *derivation graph*, as shown in Figure 9.2. In this representation, boxed nodes denote *local ambiguities*, where some segment of the input string can be derived from the same nonterminal in more than one way. A derivation graph has two advantages over representing each derivation tree separately. First, common subtrees are represented only once. Second, local ambiguities are represented explicitly in a single node, making them easy to detect and reducing the amount of space needed for their representation.

A derivation graph can be input to a disambiguation function to extract the single "right" derivation tree. For ROSIE, disambiguation is based on the order of productions in the source grammar, where productions appearing earlier in the grammar have priority over later productions. ROSIE's disambiguation function does a top-down traversal of the graph. When it encounters a local ambiguity, it compares the productions used, selects the derivation using the highest priority production, and discards the others. It then continues down the children of the selected derivation, looking for more local ambiguities to resolve. On its way back up, the disambiguation function pieces the selected derivations back together into a single derivation tree and returns that tree as its value.

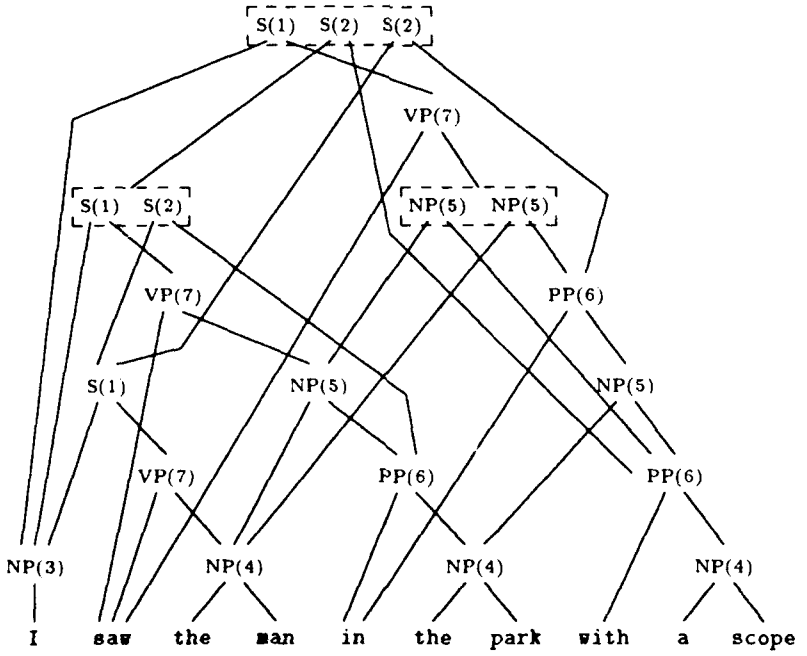


Fig. 9.2—A derivation graph

To illustrate, consider applying this disambiguation strategy to the graph in Figure 9.2. The first local ambiguity

S(1) S(2) S(2)

appears at the root of the graph. The disambiguation function would select the derivation of S(1) because Production 1 has a higher priority than Production 2. Discarding the other two derivations leaves us with the graph shown in Figure 9.3, which the disambiguation function continues to traverse until it finds the next ambiguity

NP(5) NP(5)

Here, both derivations use the same production. To resolve this ambiguity, we must call the disambiguation function recursively to find the "best" derivation tree for each derivation subgraph, and then do a left-to-right comparison of these trees, selecting the tree that uses the highest priority production at the highest level. In this instance, we would compare NP(5), the leftmost child of the first derivation, to NP(4), the leftmost child of the second. Because Production 4 has a higher priority than Production 5, we would select the second of the ambiguous derivations as "right." Discarding the first gives us the derivation tree shown in Figure 9.4, which the disambiguation function eventually returns as its selection.

A variation on this technique that seems at first glance to be better is to resolve ambiguities *as soon* as they are detected at parse time. In this way, the

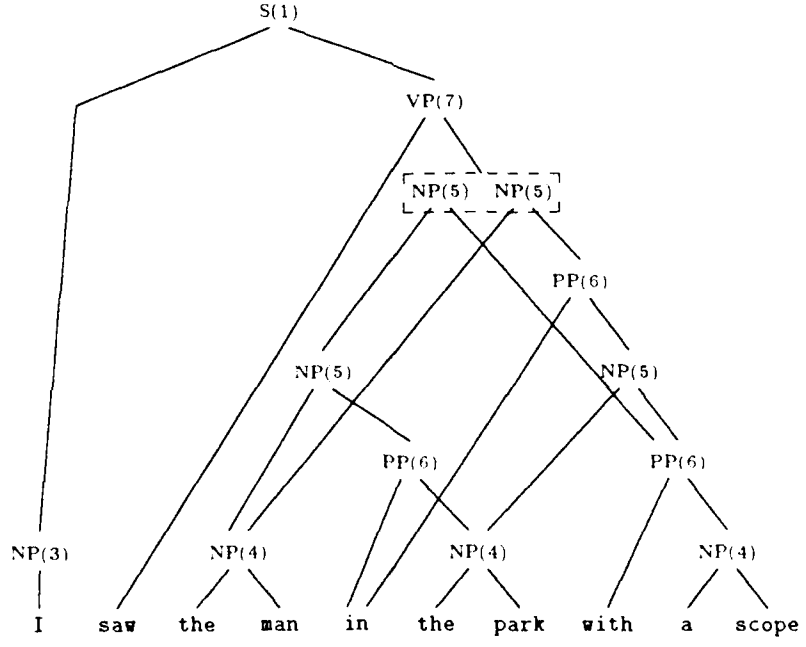


Fig. 9.3—First ambiguity resolved

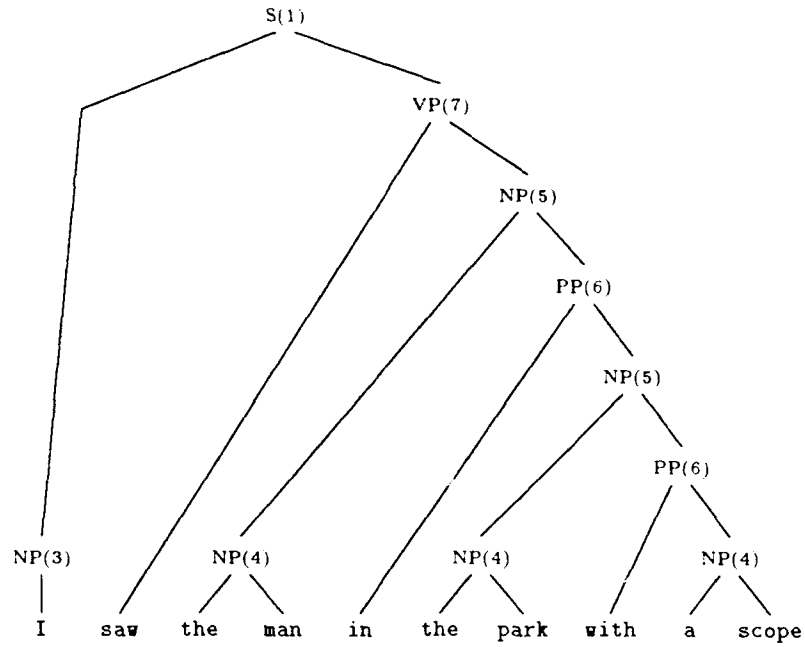
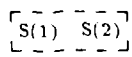


Fig. 9.4—Second ambiguity resolved

parser only outputs a single derivation tree, thus requiring less storage. However, this approach is not as time-efficient as the one outlined above, because it will try to resolve all ambiguities. By waiting until the parser is finished, we avoid resolving ambiguities that are discarded by the parser because they lie along dead-end paths. Also, by doing a top-down disambiguation of the derivation graph, we avoid resolving ambiguities in discarded derivations. For instance, we never had to resolve the ambiguity



Finally, by calling the disambiguation function on the entire derivation graph, context-sensitive techniques could be used, i.e., contextual information could be passed as an additional parameter to the disambiguation function and then used in extracting the "best" derivation.

As a further example, consider the portion of a ROSIE-like grammar shown in Figure 9.5. The principal programming statement in ROSIE is a rule, which consists of

```
(1)    <rule>  → <block> .
(2)   <block>  → <action>
(3)   <block>  → <action> AND <block>
(4)  <action>  → IF <condition> THEN <block>
(5)  <action>  → IF <condition> THEN <block> ELSE <block>
```

Fig. 9.5—Example ROSIE-like grammar

a block of one or more actions separated by the reserved word AND and terminated by a period '.'; one type of action is an if-then-else. Given this grammar, the sentential form

action AND IF *condition* THEN IF *condition* THEN *action* ELSE *action* AND *action* .

can have one of four interpretations, as shown in Figure 9.6. Of these, Interpretation *(a)* is the one normally found in modern programming languages, i.e., actions bind to the deepest action block and an else binds to the deepest if-then. By ordering the productions of the grammar as shown in Figure 9.5, the derivation tree corresponding to Interpretation *(a)* will be returned by ROSIE's disambiguation function.

As an aside, this example also illustrates a situation that, when recognized, can be used to improve the efficiency of the parser. Consider the sentential form

action AND IF *condition* THEN $action_1$ AND $action_2$ AND $action_3$ AND $\cdots$ $action_n$.

which has the n interpretations shown in Figure 9.7.

Again, the correct interpretation is *(a)*. The parser makes all n derivations because, on scanning the 2nd through nth AND, it reduces using Productions 2 or 3, deriving the action block of the if-then action. The parser reduces to a block on scanning AND because an action can be followed by an AND. But, because a block can be the last component of an action, a block can be followed by an AND as well. However, we know that with our disambiguation function an action block followed by an '**AND** $action_i$' phrase will always be discarded in favor of a longer action block that includes that phrase. If we remove the action to reduce using Productions 2 or 3 upon scanning an AND, the parser will return a

```
action AND
IF condition
 THEN IF condition
       THEN action
       ELSE action AND
            action .
```

(a)

```
action AND
IF condition
 THEN IF condition
       THEN action
       ELSE action AND
action .
```

(b)

```
action AND
IF condition
 THEN IF condition
        THEN action
 ELSE action AND
      action .
```

(c)

```
action AND
IF condition
 THEN IF condition
        THEN action
 ELSE action AND
action .
```

(d)

Fig. 9.6—Four interpretations

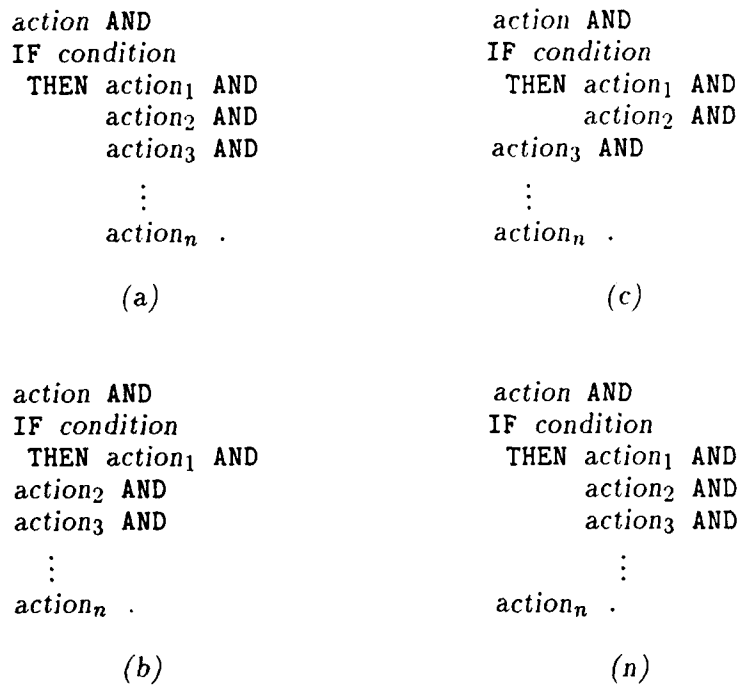


Fig. 9.7—n interpretations

single, unambiguous derivation tree corresponding to Interpretation (a) and it will not have executed $n - 1$ spurious reduce actions.

To use this observation to optimize ROSIE's parser, we examined the grammar looking for productions that would generate spurious reduce actions and then made declarations to the parse table constructor that would cause it not to generate those actions. Although this approach helped to improve the parser's run-time performance, it has not been generalized into an automatic optimization technique.

X. CONCLUSION

The parsing system outlined in the preceding sections has gone through several years of evolution since its initial development for ROSIE. The parser currently being distributed with ROSIE (Version 3.0) (Kipps et al., 1987) is implemented in Portable Standard LISP (PSL) (Galway et al., 1984), but is not up-to-date with the parsing system described in this Note. Essentially, the disambiguation and code generation functions operate as described; the parser and parse table constructor are substantially different in appearance but are similar in much of their basic operation. The version described here has been implemented in Common LISP (Steele, 1984).

To make the advanced parsing technology used by ROSIE available to a wider range of language development projects, the future objective for this work is the development of a "next-generation" YACC-like set of tools. YACC (Johnson, 1975) is a compiler-compiler that can recognize a subset of context-free languages and allows for a small degree of ambiguity. YACC is not capable of generating a compiler for ROSIE, but a similar tool using Tomita's algorithm would have reduced the level of effort in developing ROSIE's parsing system from man-years to a man-month or less. In addition, if this tool were implemented in C as opposed to LISP, significant performance improvements could be expected.

As others begin to extend the expressiveness and readability of new high-level programming languages, or as they try to improve the level of dialogue available in user front-ends, they will likewise find that they require the power of general context-free parsing. Tomita's algorithm provides a practical approach to meeting this need.

Appendix A

CODE GENERATION

For ROSIE, the parsing system is used to translate source programs into evaluable LISP expressions. Each production in a ROSIE grammar file has the form

$$(D_p \ C_{p1}C_{p2}\cdots C_{p\bar{p}} \ \texttt{->} \ template)$$

where *template* is the component of the production that describes the LISP expression generated by it. For example, a production in standard BNF notation (Pagan, 1981), such as

```
<condition> ::= IF <expression> THEN <series> ELSE <series>
```

would be described in the bnf file as

```
(<condition> IF <expression> THEN <series> ELSE <series>
   -> (COND ($1 !$2) (T !$3)))
```

and used for translating if-then-else statements of the source language into COND statements of LISP.

Code Generation

ROSIE's code generation routine operates by doing a preorder traversal of the derivation tree returned by disambiguation. The resulting LISP expression is generated in a left-to-right manner from the bottom up. At each internal node of the tree (denoting a derivation based on Production p) the routine first generates the subexpressions for the node's children, i.e., corresponding to the derivations for $C_{p1},\cdots,C_{p\bar{p}}$. If C_{pi} is a terminal, then no subexpression is generated for it; if a preterminal, then the subexpression is the matching input token; and if a nonterminal, then the subexpression is computed by applying the code generation routine to the corresponding child.

After the subexpressions are generated, they are bound to the variables $\$1,\cdots,\$q$, respectively, where q equals the number of preterminals and nonterminals in $C_{p1},\cdots,C_{p\bar{p}}$. $\$j$ is bound to the subexpression generated for the jth preterminal or nonterminal. Finally, referring to these variables, the template of Production p specifies the form of the LISP expression to be generated for the node.

Template Semantics

The template of a production specifies the expression generated for the portion of the derivation tree representing an instance of that production. All or part of a template can be used as a literal; some portions can be replaced by subexpressions generated by nonterminals and preterminals in the right-hand side of the production, while others can be replaced by the results of computations specified in the template.

For every variable (**$***j*) in the template, the resulting expression will find that variable replaced by the subexpression bound to it. If the variable is immediately preceded by an exclamation point (**!$***j*), then the subexpression must be a list and is spliced into the resulting expression. For example, a list of one or more numbers (where ***number** is a preterminal matching any number) can be described by the two productions

```
(<numlist> *number -> ($1))
(<numlist> *number , <numlist> -> ($1 !$2))
```

In generating an expression for a recognized instance of the first production, **$1** is bound to the subexpression generated for ***number**, which in the case of a preterminal is the token (a number) matching it. The expression then generated is a list containing only this subexpression. In generating an expression for the second production, **$1** will again be bound to the subexpression generated for ***number**, while **$2** will be bound to the subexpression generated for **<numlist>**, which will be a list of one or more numbers. The expression generated by this production is a list containing the number bound to **$1** as its first element with the remaining elements of the list taken from **$2**—this is equivalent to **(cons $1 $2)** in LISP. Thus, the expression generated for **1,2,3** would be **(1 2 3)**.

In order to allow more complex semantics, a hook is provided to evaluate LISP expressions at code generation time and then substitute or splice in the resulting value. If the first element of a list in the template is the symbol **$eval**, then the second element of the list must be an expression, which will be evaluated at code generation time, i.e.,

($eval *expression*)

This expression may use the variables $1,$\cdots$,**$***j*. The value returned by evaluating the expression will replace the **$eval** form in the expression generated from the template. If the **$eval** form is preceded by an exclamation point, i.e.,

!($eval *expression*)

the resulting value will be spliced into the generated expression. For example, the two productions seen earlier could be specified as

```
(<numlist> *number -> ($eval (list $1)))
(<numlist> *number , <numlist> -> ($eval (cons $1 $2)))
```

and generate the same expression. We could also specify the productions

```
(<sumlist> *number -> $1)
(<sumlist> *number , <sumlist> -> ($eval (+ $1 $2)))
```

and simply generate the cumulative total of the numbers in the list. More important, **$eval** forms can be used to check context-sensitive aspects of a grammar (e.g., a list of numbers in ascending order) or to make complex transformations on the expressions being generated.

A null production is simply specified by excluding the right-hand side of the production. For example, the production

```
(<sumlist> -> 0)
```

denotes that an empty sumlist has a value of 0. If no template is specified for a production, e.g.,

```
(<sumlist> *number)
```

then a default template of the form

-> ($1 $\cdots$ $n)

is automatically generated for it, where n is the number of nonterminals and preterminals in the right-hand side of the production. For null productions, the default template is the empty list, i.e.,

```
-> ( )
```

If the template contains the symbol **$tokens**, it will be replaced in the generated expression by the list of tokens recognized by that instance of the production. For example, given the production

```
(<savelist> <sumlist> -> ($1 $tokens))
```

the expression generated for `1,2,3` would be `(6 (1 , 2 , 3))`. The symbol **!$tokens** splices the list of tokens into the expression.

Appendix B

THE CONSTRUCTOR

Tomita's parsing algorithm utilizes a minor variation on standard LR parse tables. Any of the several parse table construction algorithms (LR(0), SLR(1), LR(1), LALR(1), etc.) could be modified to produce the necessary parse table. The only modification required is simply to allow each entry in the action table to record a *set* of actions rather than a single action.

The Extended LR Parse Table

Briefly, to modify a standard constructor, locate the portion of the algorithm that detects and deals with *shift-reduce* conflicts generated when the algorithm goes to record an action in a table entry for which an action already exists. Typically when such conflicts occur, the constructor either halts and reports an error or records only one action and gives a warning. For Tomita's algorithm, the constructor should record all entries as a set of actions and add all conflicting actions to the set. For any context-free grammar, each set can contain at most one shift or accept action (never both), and any number of reduce actions.

The parse table construction algorithm described below is a variation on the SLR (Simple LR) constructor found in Aho and Ullman (1977). The central concepts of this approach are (1) to construct a *deterministic finite automata* (DFA) that recognizes *viable prefixes* of a grammar, and (2) to convert the states and transition function of the DFA into the states and actions of an extended LR parse table.

Items and Sets of Items

The first step in constructing an extended LR parse table is the construction of a DFA that recognizes the *viable prefixes* of the source grammar. A viable prefix is a prefix of a sentential form, i.e., if α is a viable prefix, then

$$R \overset{*}{\Rightarrow} \alpha\beta$$

for some β. A viable prefix is so named, because it is possible to add terminal symbols to the end of a viable prefix to complete a sentential form. As a result, we know that there is no apparent error as long as the portion of the input string scanned to a given point can be reduced to a viable prefix.

The algorithm for constructing the DFA is couched in terms of *items* and *sets of items*. We define an *LR(0) item* (or *item* for short) of a grammar G to be a production of G with a marker '∘' appearing somewhere in its right-hand side. For example, the production $\mathtt{A} \rightarrow XYZ$ generates the four items

$$\begin{aligned} \mathtt{A} &\rightarrow \circ XYZ \\ \mathtt{A} &\rightarrow X \circ YZ \\ \mathtt{A} &\rightarrow XY \circ Z \\ \mathtt{A} &\rightarrow XYZ \circ \end{aligned}$$

Null productions, such as $\mathtt{A} \rightarrow \epsilon$, generate a single item, $\mathtt{A} \rightarrow \circ$.

An item acts as a placeholder, indicating how much of a production has been recognized at a given point in the parsing process. Thus, the item

$$\mathtt{A} \rightarrow \alpha \circ \beta$$

denotes the situation in which the parser has scanned input corresponding to the symbols in α and expects the next part of the input to correspond to β. To be more specific, after scanning the ith symbol of input string $\mathtt{x}_1 \cdots \mathtt{x}_n$, then, for some j $(1 \leq j \leq i)$ and some m $(i+1 \leq m \leq n)$, if

$$\alpha \stackrel{*}{\Rightarrow} \mathtt{x}_j \cdots \mathtt{x}_i$$

then we expect

$$\beta \stackrel{*}{\Rightarrow} \mathtt{x}_i + 1 \cdots \mathtt{x}_m$$

Items in which β is empty (e.g., $\mathtt{A} \rightarrow XYZ\circ$) denote the situation in which an instance of the production has been successfully recognized in the input symbols just scanned.

Where an item denotes the partial recognition of some production, a *set of items* extends this concept to several possible partial recognitions, all of which are consistent with the input scanned so far. Individually, items can be viewed as the states of a *nondeterministic finite automata* (NFA) for recognizing viable prefixes. Grouping items together into sets is similar to the process of constructing a DFA from the NFA. These sets of items later give rise to the states of the parse table.

Constructing the Canonical Collection of LR(0) Items

The set of items used to construct the parse table for a class of LR parsers known as "simple" LR (SLR) is called the *canonical collection of LR(0) items*. This set is sufficient for building a parse table to be utilized by Tomita's algorithm. Its construction is described below.

Notation. Number the productions of grammar G arbitrarily $1, \cdots, d$, where each production is of the form

$$D_p \rightarrow C_{p1} \cdots C_{p\tilde{p}} \quad (1 \leq p \leq d)$$

and where $\tilde{p}$ is the number of symbols in the right-hand side of the pth production. Augment G with a 0th production

$$D_0 \rightarrow R \dashv$$

where R is the root of G and '$\dashv$' is the end-of-sentence symbol. The purpose of this new production is to indicate to the parser when it has reached an accept state; the accept action corresponds to shift action on the end-of-sentence symbol.

Definition. A set of items (I) gives rise to successor sets and transitions between sets using the functions CLOSURE and GOTO.

- CLOSURE(I) returns a set of items constructed from I by the following rules.
  (1) Every item in I is in CLOSURE(I).
  (2) If $\mathtt{A} \rightarrow \alpha \circ \mathtt{B}\beta$ is in CLOSURE(I), then for each production $\mathtt{B} \rightarrow \gamma$, add the item $\mathtt{B} \rightarrow \circ\gamma$ to CLOSURE(I), if it is not already there.

  Intuitively, $\mathtt{A} \rightarrow \alpha \circ \mathtt{B}\beta$ in CLOSURE(I) indicates that symbols derivable from $\mathtt{B}\beta$ may be expected to appear next in the input string. It follows that for each production $\mathtt{B} \rightarrow \gamma$, we may also expect to see symbols derivable from γ. For this reason, the items $\mathtt{B} \rightarrow \circ\gamma$ are added to CLOSURE(I). The set of items computed by CLOSURE(I) is completed, or *closed*, when no new items can be added to it.
- GOTO(I,X), where I is a closed set of items and X is a vocabulary symbol of G (i.e., a terminal, preterminal, or nonterminal), returns the closure of the set of all items $\mathtt{A} \rightarrow \alpha X \circ \beta$ such that a subset of items $\mathtt{A} \rightarrow \alpha \circ X\beta$ exists in I. Intuitively, GOTO establishes links between closed sets of items, corresponding to transitions between states of a DFA, i.e., GOTO(I,X) defines a transition labeled X from I to the set of items returned by GOTO(I,X).

Each state of the LR parse table is derived from a distinct set of items (I). The set I_0, corresponding to the start state S_0, is closed over the single item

$$D_0 \rightarrow \circ R \dashv$$

denoting that, at the start of a parse, no symbols have been scanned but we expect to see a sentence of the language defined by G.

Definition. The canonical collection of sets of LR(0) items for a grammar G′ (i.e., grammar G, augmented by the 0th production $D_0 \rightarrow R \dashv$) is constructed with the function ITEMS(G′).

- ITEMS(G′) constructs and returns C, the canonical collection of sets of items, in the following manner.

  (1) C and C' are initialized to be sets containing the single set of items I_0, the closure of

  $$\{D_0 \rightarrow \circ R \dashv\}$$

  Repeating until there are no more sets of items in C',

  (2) A set of items I is removed from C'.

  (3) For each vocabulary symbol X in G, the set of items generated from GOTO(I,X) is added to C and C', unless it already exists in C.

  Once completed, the sets of items C returned by ITEMS corresponds to the states of a DFA recognizing the viable prefixes of grammar G′, while the GOTO function for this set of items, referred from here on as GOTO_C, corresponds to the transition function of that DFA, mapping state to states.

The functions CLOSURE, GOTO, and ITEMS are defined in Figure B.1. The canonical set of items generated for the grammar of Figure 5.1 is shown in Figure B.2. The GOTO function for this set of items is shown as a transition diagram of a DFA in Figure B.3.

```
ITEMS(G′)
 Let C := {CLOSURE({D0 → ∘R⊣})}
 For each I ∈ C,
   for each distinct X | A → α∘Xβ ∈ I,
     let C := C ∪ {GOTO(I,X)}
 Return(C)

CLOSURE(I)
 For each ⟨p,j⟩ ∈ I | j ≠ p̄,
   if C_p(j+1) ∈ N,
     for each D_q ∈ N | C_p(j+1) = D_q,
       let I := I ∪ {⟨q,0⟩}
 Return(I)

GOTO(I,X)
 CLOSURE(⋃_{A→α∘Xβ∈I} A → αX∘β
```

Fig. B.1—Constructing the canonical set of LR(0) items

I_0:
D$_0$ → ∘S ⊣
S → ∘NP VP
S → ∘S PP
NP → ∘*n
NP → ∘*det *n
NP → ∘NP PP

I_1:
D$_0$ → S∘⊣
S → S∘PP
PP → ∘*prep NP

I_{acc}:
D$_0$ → S ⊣∘

I_2:
PP → *prep∘NP
NP → ∘*n
NP → ∘*det *n
NP → ∘NP PP

I_3:
PP → *prep NP∘
NP → NP∘PP
PP → ∘*prep NP

I_4:
NP → NP PP∘

I_5:
NP → *det∘*n

I_6:
NP → *det *n∘

I_7:
NP → *n∘

I_8:
S → S PP∘

I_9:
S → NP∘VP
NP → NP∘PP
VP → ∘*v NP
PP → ∘*prep NP

I_{10}:
VP → *v∘NP
NP → ∘*n
NP → ∘*det *n
NP → ∘NP PP

I_{11}:
VP → *v NP∘
NP → NP∘PP
PP → ∘*prep NP

I_{12}:
S → NP VP∘

Fig. B.2—Canonical LR(0) collection

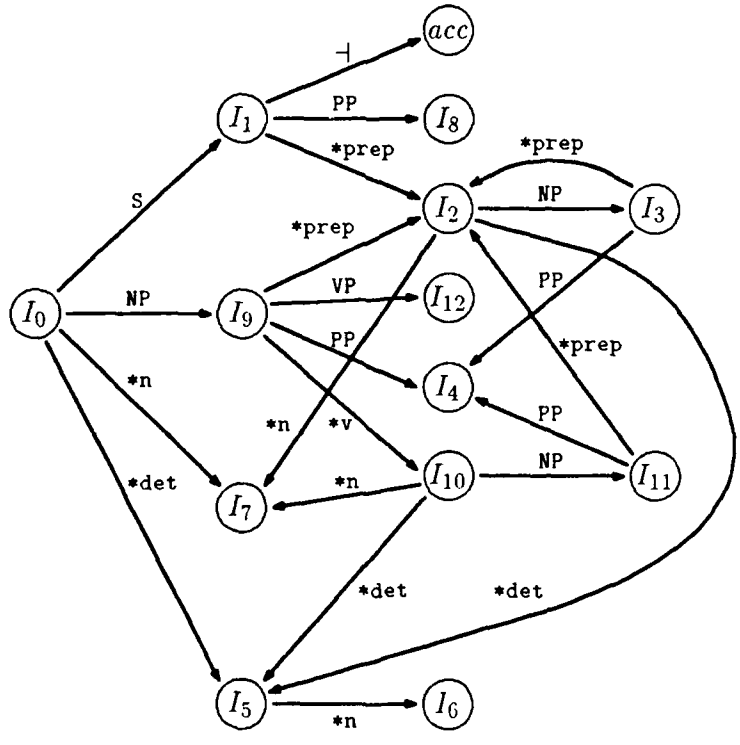


Fig. B.3—GOTO$_C$ graph

Constructing the Extended LR Parse Table

Given the canonical set of items C denoting a DFA that recognizes viable prefixes of G, we next construct an extended (multi-action) LR parse table from C in terms of the ACTIONS and GOTO functions called by the parser. These functions are constructed using a modification of the SLR parse table construction technique, as described below.

Definition. To determine reduce actions, the construction algorithm requires the functions FIRST and FOLLOW.

- FIRST(α) returns the set of terminals and preterminals that can begin a string derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also returned by FIRST(α).

  To compute FIRST(X) for all vocabulary symbols X (i.e., strings α of length 1), apply the following rules until nothing more can be added to any FIRST(X) set.

  (1) If X is a terminal or preterminal, then FIRST(X) is $\{X\}$.

  (2) If X is a nonterminal, then for each production $X \rightarrow Y\alpha$ such that Y is a terminal or preterminal, add Y to FIRST(X). If $X \rightarrow \epsilon$ is a production, add ϵ to FIRST(X).

  (3) In general, if X is a nonterminal, then for each production

  $$X \rightarrow C_{p1}C_{p2}\cdots C_{p\bar{p}}$$

  and some i such that all of $C_{p1},\cdots,C_{pi-1}$ are nonterminals and FIRST(C_{pj}) (for $j = 1,\cdots,i-1$) contains ϵ (i.e., $C_{p1},\cdots,C_{pi-1} \stackrel{*}{\Rightarrow} \epsilon$), add every non-$\epsilon$ symbol in FIRST(C_{pk}) (for $k = 1,\cdots,i$) to FIRST(X). If ϵ is in FIRST(C_{pj}) for all $j = 1,\cdots,\bar{p}$, then add ϵ to FIRST(X).

  Similarly, to compute FIRST(α) for any string of vocabulary symbols $\alpha = X_1\cdots X_n$ apply the rules:

  (4) If X_1 is a terminal or preterminal, then FIRST($X_1\cdots X_n$) is $\{X_1\}$;

  (5) For some i such that $X_1,\cdots,X_{i-1}$ are nonterminals and FIRST(X_j) (for $j = 1,\cdots,X_{i-1}$) contains ϵ, add every non-ϵ symbol in FIRST(X_k) (for $k = 1,\cdots,i$) to FIRST($X_1\cdots X_n$). If ϵ is in FIRST(X_j) for all $j = 1,\cdots,n$, then add ϵ to FIRST($X_1\cdots X_n$).

- FOLLOW(A) returns the set of terminals and preterminals that can appear immediately to the right of nonterminal A in some sentential form of G'. That is, the set of terminals and preterminals

  $$\{X \mid D_0 \stackrel{*}{\Rightarrow} \alpha A X \beta\}$$

  If A can be the rightmost symbol in some sentential form of G, then the end-of-sentence symbol '⊣' appears in FOLLOW(A).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW(A) set.

(1) The end-of-sentence symbol '⊣' is in FOLLOW(R), where R is the root of G.

(2) For each production A $\rightarrow \alpha$Bβ, such that β is not empty, add everything in FIRST(β), except ϵ, to FOLLOW(B).

(3) For each production A $\rightarrow \alpha$Bβ, such that β is empty or FIRST(β) contains ϵ, add everything in FOLLOW(A) to FOLLOW(B).

The FIRST and FOLLOW tables computed for the grammar in Figure 5.1 are shown in Figure B.4.

Notation. Let $C = \{I_0, I_1, \cdots, I_n\}$ be the canonical set of LR(0) items constructed for grammar G', and let GOTO$_C(I,X)$ be the associated transition function. The states of the parse table are $S_0, S_1, \cdots, S_n$, where state S_i is constructed from I_i.

The Constructor. The constructor is a function of one argument CONST(C). CONST determines the entries of the action and goto tables for each state S_i using the following rules.

(1) For all items $D_p \rightarrow \alpha \circ X\beta$ in I_i such that GOTO$_C(I_i,X) = I_j$, if X is a terminal or preterminal, add sh$j$ to ACTIONS(S_i,X), otherwise set GOTO(S_i,X) to j.

(2) For all items $D_p \rightarrow \alpha \circ$ in I_i, where $p \neq 0$, add re$p$ to ACTIONS(S_i,X) for all X in FOLLOW(D_p).

(3) If item $D_0 \rightarrow R \dashv \circ$ is in I_i, then add acc to ACTIONS(S_i,⊣).

(4) All entries not defined by rules (1) through (3) are made errors.

The action and goto tables of the grammar in Figure 5.1 appear in Figure 5.2.

A Space-Efficient Implementation of the Constructor

A space-efficient algorithm for constructing an extended LR parse table appears in Figure B.5. Figure B.6 defines the ACTIONS and GOTO functions of the parser in terms of this table representation.

X	FIRST(X)	FOLLOW(X)
S	{*det,*n}	{⊣,*prep}
NP	{*det,*n}	{⊣,*prep,*v}
VP	{*v}	{⊣,*prep}
PP	{*prep}	{⊣,*prep,*v}

Fig. B.4—FIRST and FOLLOW tables

In CONST, [1] initializes C, the set of item set/state pairs, and then constructs the start state corresponding to the initial item $\langle 0, 0 \rangle$.

In GET-STATE, [2] checks if a state of item set I has not already been recorded in C, returning that state if it has and constructing that state otherwise.

In CONST-STATE, [3] initializes the action and goto tables of a new state S to be associated with I in C; T' is a temporary table used to record shift and goto actions; [4] iterates over each item in the closure of I. If an item indicates the complete recognition of production p, [5] enters a reduce action in the action table under each terminal and preterminal that can follow nonterminal D_p; otherwise, there will be a transition across the $j+1$st symbol of production p into the item set I'. This transition will result in a shift action or a goto and [6] records it in the temporary transition table T'. After recording all reduce actions and noting all transitions, [7] iterates through the transitions in T'. For each transition into the item set I', [8] accesses the state S' associated with I'. If the transition is across a nonterminal, [9] makes the appropriate entry into the goto table; otherwise, [10] adds a shift action to the reductions already recorded in the action table; finally, [11] returns the newly constructed state.

```
     CONST(G')
[1]   Let C be empty
      Return(CONST-STATE({⟨0,0⟩}))

     GET-STATE(I)
[2]   If ∃⟨I,s⟩ ∈ C,
        return(s)
      else
        return(CONST-STATE(I))

     CONST-STATE(I)
[3]   Let T_action, T_goto and T' be empty
      Let s := ⟨T_action, T_goto⟩
      Let C := C ∪ {⟨I,s⟩}
[4]   For each ⟨p,j⟩ ∈ CLOSURE(I),
[5]     if j = p̄,
          for each X ∈ FOLLOW(D_p)
            if ∃⟨X,a⟩ ∈ T_action,
              let a := a ∪ {'re p'}
            else
              let T_action := T_action ∪ {⟨X,{'re p'}⟩}
        else
[6]       if ∃⟨C_p(j+1), I'⟩ ∈ T'
            let I' := I' ∪ {⟨p,j+1⟩}
          else
            let T' := T' ∪ {⟨C_p(j+1), {⟨p,j+1⟩}⟩},
[7]   For each ⟨X,I'⟩ ∈ T',
[8]     let s' := GET-STATE(I')
[9]     if X ∈ N,
          let T_goto := T_goto ∪ {⟨X,s'⟩}
        else
[10]      if ∃⟨X,a⟩ ∈ T_action,
            let a := a ∪ {'sh s'}
          else
            let T_action := T_action ∪ {⟨X,{'sh s'}⟩}
[11]  Return(s)
```

Fig. B.5—The constructor

```
ACTIONS(s = ⟨T_action, T_goto⟩, x)
 If ∃⟨x,a⟩ ∈ T_action,
   return(a)
 else
   return({ })

GOTO(s = ⟨T_action, T_goto⟩, D_p)
 Return('s') | ⟨D_p, s'⟩ ∈ T_goto
```

Fig. B.6—The ACTION and GOTO functions

REFERENCES

Aho, A.V., J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

Aho, A.V., J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

Backus, J.W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," in *Proceedings of the International Conference on Information Processing*, UNESCO, 1959, pp. 125-132.

Chomsky, N., "Three Models for the Description of Language," in *IRE Transactions on Information Theory*, Vol. 2, No. 3, 1956, pp. 113-124.

Earley, J., *An Efficient Context-Free Parsing Algorithm*, Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1968.

Earley, J., "An Efficient Context-Free Parsing Algorithm," in *Communications of the ACM*, Vol. 13, No. 2, Feb. 1970, pp. 94-102.

Irons, E.T., *Syntax Graphs and Fast Context-Free Parsing*, Research Report No. 71-1, Department of Computer Science, Yale University, New Haven, CT, Jan. 1971.

Johnson, S.C., "YACC—Yet Another Compiler Compiler," CSTR 32, Bell Laboratories, Murray Hill, NJ, 1975.

Ginsburg, S., J.S. Ullian, "Preservation of Unambiguity and Inherent Ambiguity in Context-Free Languages," in *Journal of the ACM*, Vol. 13, No. 3, 1966, pp. 364-368.

Galway, W., M.L. Griss, B. Morrison, B. Othmer, *The Portable Standard LISP User Manual*, The Utah Symbolic Computation Group, University of Utah, Salt Lake City, UT, 1984.

Kipps, J.R., B.A. Florman, H.A. Sowizral, *The New ROSIE Reference Manual and User's Guide*, The RAND Corporation, R-3448-DARPA/RC, June 1987.

Kipps, J.R., *Analysis of a Table-Driven Algorithm for Fast Context-Free Parsing*, The RAND Corporation, P-7452, June 1988.

Knuth, D.E., "On the Translation of Languages from Left to Right," in *Information and Control*, Vol. 8, 1965, pp. 607-639.

Pagan, F.G., *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Steele, G.L., *Common LISP: The Language*, Digital Press, Bedford, MA, 1984.

Tomita, M., *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*, Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1985a.

Tomita, M., "An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications," in *Proceedings of the Ninth IJCAI*, Vol. 2, Los Angeles, CA, Aug. 1985b, pp. 756-764.

Younger, D.H., "Recognition and Parsing of Context-Free Languages in Time n^3," in *Information and Control*, Vol. 10, No. 2, 1967, pp. 189-208.